UiO : Department of Informatics
University of Oslo

IDS-based Passive Asset Detection

Using and extending an IDS for asset detection

Philip Christian Scheel

Master's Thesis Autumn 2014

IDS-based Passive Asset Detection

Philip Christian Scheel

15th August 2014

Abstract

Computer networks are growing in complexity and size, making it challenging to keep an inventory of computers and their software operating in a network. Traditional solutions to this problem, like active scanning or software agents on individual hosts cannot function in all environments and can incur traffic overhead in the network or processing overhead on the network hosts - and are therefore not the best solution in all environments.

This Master's project has investigated the potential of using the power of Intrusion Detection Systems, more specifically the IDS Bro, for asset detection. The resulting system is passive, and therefore does not incur any network traffic overhead, nor does it incur any processing overhead on the monitored hosts. The thesis investigates techniques for gathering fingerprints relevant to an IDS asset detection system, and describes a proof of concept implementation to demonstrate the usefulness of this method.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisors, Professor Audun Jøsang, Tom Danielsen and Stian Jahr for their feedback, corrections, suggestions and guidance on my thesis. Your help was invaluable to my work.
I would also like to thank André Whitehouse for his help reading and commenting on this thesis.
I would like to thank my colleagues at *mnemonic* for providing a friendly, supportive work environment and for generously sharing their knowledge, which has helped me develop professionally these last 3 years.
And finally, I would like to thank my family and friends for their support for the duration of this project.

For all you have done for me, you have my sincere thanks.

Philip Christian Scheel
University of Oslo
August, 2014

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and motivation

With the rapid growth in both internet-connected devices and increased reliance on networked technology for businesses, it is becoming increasingly important to adequately protect assets from adversaries in the form of hostile agents and malicious software.

As networks grow in size and their connected hosts increase in complexity, the task of keeping track of assets - such as physical machines and their software - also increase in complexity. While smaller organizations can keep track of assets manually, larger companies will eventually require automation or improved processes to keep an accurate overview of their assets.

Solutions such as host-based agents which report software installations are typically impractical when those in need of this data are not in control of the devices where these assets resides. Active scanning using tools such as NMAP will only detect software listening on ports it can connect to. Additionally, these scans will have to be continuously maintained in case of changing configurations of the surveilled hosts.

Passive asset detection generally consists of one or more centrally placed sensors placed in strategic places in networks where such detection is desirable. Traffic is mirrored out from these points, and the asset detection software will categorize the traffic and infer information about the assets in the network. Because the traffic is captured in its entirety, an asset detection system can evaluate relevant data from the lowest levels of the

Internet Protocol model (such as MAC addresses) up to software-specific data in the application layer.

Bro is a network intrusion detection system with an event-based scripting language that allows reasoning over the data gathered. The software can be installed on Linux, FreeBSD, and Mac OS X based hosts. Owing to built-in parsers for many protocols, it is often simple to extract relevant data and create scripts that trigger on malicious traffic or on other forms of relevant information. In this Master's project, Bro has been used as a platform to implement asset detection methods.

1.2 Goals and research questions

The main goal of this thesis is to investigate a methodology for passive asset detection on higher levels of the OSI model, and making a proof of concept implementation based on this methodology.

The system should be relatively easy to extend for someone versed in programming. Our fingerprinting-scripts should minimize the amount of false positives, and not only detect ports used for services, but also the software listening on the port. Client side software should also be detected.

The implemented system will consist of four parts:

- A Bro IDS installation
- A set of signature-scripts written for Bro to detect a limited set of asset-related information
- A program that collects and inserts asset information into a database
- A system to present the data in an accessible manner

Based on these goals the research questions for this thesis has been stated as follows:

- Q1) How can we design an asset detection system based on an IDS, such as Bro?
- Q2) How practical is the process of collecting and using fingerprints in an IDS context?
- Q3) What is the practical possible coverage of this type of system?

1.3 Outline

The thesis is divided into 6 chapters. Chapter 2 explains the background for asset detection, and provides most of the necessary background for this thesis. Chapter 3 describes the design of the system used for the project, along with tools and techniques used for gathering the data required for writing signatures for detecting assets.

Chapter 4 describes the implementation of these techniques in making specific signatures, demonstrating what information can be gathered at different stages of software's lifecycle, and what data can be extracted from it.

Chapter 5 evaluates and discusses the results of implementing the thesis. Chapter 6 discusses the results, and some of the potential problems for a passive asset detection system. Chapter 7 concludes the thesis, and suggests future work and possible tools for asset detection systems.

Chapter 2

Background

In this chapter, we discuss the background of asset detection, as well as the motivations behind our target system. The chapter ends with a discussion of some threats against asset detection and introduces the concept of a software lifecycle.

2.1 What is asset detection?

To understand the concept of asset detection, we must first understand what an asset is. The ISO-standard ISO 13335-1:2004 (section 3.2) defines several types of assets, among them "physical assets (e.g., computer hardware, communications facilities, buildings), information / data (e.g., documents, databases), software (...)". These are assets that can fall to IT departments to track, analyze and protect, which makes it a worthwhile goal to have a way to detect and organize this information.

Asset detection is this requirement set into practice. Asset detection is a process of – usually - continuous tracking of assets in an organizations inventory.

In an ideal situation, an asset management system will have access to a wide variety of data relevant to the needs of an administrator or other users of the system. Data such as that mentioned in the ISO definition, but also potentially things like the hostname of each machine, users operating it and other metadata that may be useful.

This thesis concentrates on the assets mentioned in the excerpt of the

ISO definition, as these are the asset types that are generally available for analysis for a network-based listening service. Passive network based asset detection only has access to data transferred over the wire, and therefore has some limitations when compared to other more active or invasive forms of asset management.

2.2 Why asset detection?

There are numerous motivations for doing asset detection in an organization. This section explains some of these.

2.2.1 Asset consolidation

Once an organization has gained an overview over its assets, it can accomplish various tasks with it.

One of these tasks is the unification of versions of software and hardware. By keeping assets uniform across the organization, much of the work that goes into repeated tasks are simplified because these tasks are done over a smaller set of variables. This simplifies tasks such updates, upgrades, software and hardware deployment, planning, troubleshooting and other use cases.

Asset detection can be used both for the initial mapping of software and hardware in an organization, as well as in later planned upgrade cycles. Many companies have a PC life cycle where machines are upgraded every few years. In these situations, an asset management system will ideally hold all the necessary information to find the machines that should be upgraded. This situation carries over to software, such as planned upgrades for operating systems or software applications.

2.2.2 Asset management

Closely tied to the arguments for consolidating software and hardware is the potential usage of statistics taken from an asset management system.

Statistics on an organization's install base allows for evidence-based choices when planning and designing future solutions.

An example of where such usage is useful is when deciding specifications for new software. In the case of the development of a custom web application, it makes sense to consult a list of web browsers currently in use in the organization, and setting down compatibility requirements according to this information.

Another potential usage is the possibility of detecting installations of software with special licensing requirements. This can allow for verification of license compliance in cases where the license allows for only specific numbers of software installations.

2.2.3 Defense against threats

A very interesting use case for asset information is the case of defense against various security threats. There are several ways one can leverage good information about assets in the organization to reduce the exposure to threats.

An attack surface as a concept in software can be summarized as the total sum of attack vectors available in an application. In the OWASP wiki [3], an attack surface is described as all of the different points where an attacker could get into a system, and where they could get data out.

A system with more than one version of the same software has at a minimum the same amount of shared attack vectors in each version, and in most cases will have additional ones based on either older vectors which have been fixed (bugs) or newer vectors added in the newer versions.

When an organization has more than one version of software available in a network, it offers an attacker more potential points to penetrate security or pivot their attack from. A visualization of the attack surface of two separate software versions can be seen in figure 2.1.

By having an overview of software in use in an organization, it is possible to reduce the attack surface, both by consolidating software versions – ideally to the newest version – and different software that serves the same purpose in an organization. In environments where the IT department is

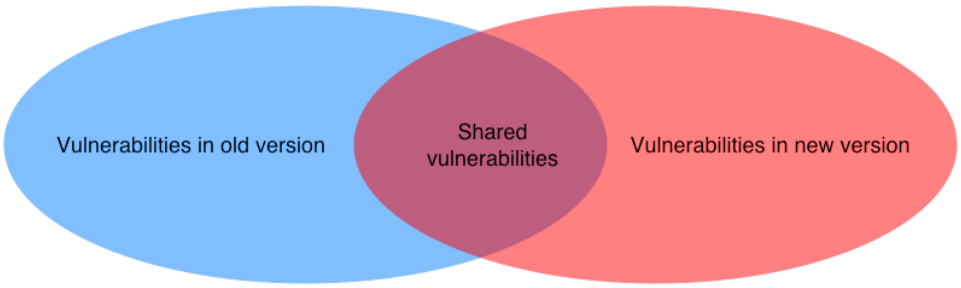


Figure 2.1: Venn diagram of minimum and maximum vulnerabilities

not able to push updates to the clients automatically, the use of software version lists allows them to contact end users to make them install required updates, or to uninstall software not allowed under the organization policy.

Another important method in the effort of securing the organization against network related threats is the binding between software versions and attack signatures through the use of Common Vulnerabilities and Exposures initiative's identifiers (CVEs)[9]. In many IDS systems, such as Snort, Suricata and ISS Proventia, many of the signatures will have one or more associated CVEs. By listing software which is in use and accessible in the organization through the network, it is possible to map out a threat profile over what vulnerabilities the network is exposed to. This can be helpful in two very meaningful ways.

When profiling the IDS sensor and policy, such a list allows an analyst to remove the signatures that cover vulnerabilities the organization is not likely to have.

Secondly, when reviewing the total set of signatures available to the sensor, finding it to not cover all vulnerabilities, new and relevant signatures can be written, or an assessment of the security risk of continued use of software with an uncovered vulnerability can be done.

Another use of asset detection system for defensive purposes is the potential of using fingerprints of files to detect what files are being served from servers under the system's protection – and when these files are modified. In the case of files being served over HTTP, there is always a risk of a compromised server distributing modified – and in many cases malicious – files to users. By tagging these files as assets and detecting their transfers and modification, we can potentially detect compromised servers and

potentially stop a loss of reputation as a consequence of compromising a customer.

A final reason why good knowledge about assets is important is the support it offers in the investigation by analysts during security incidents. When an IDS alarm has been triggered, one of the first steps to an analyst is to decide if the event is a false positive or not. If the attack is specific to a vulnerability, and the victim machine is not running software which has that vulnerability, it is likely that the attack has not been successful. By having information that allows an analyst to make this conclusion early, and thus discount the event, the analyst is freed to spend more time on events that are likely to be impactful.

2.3 Existing approaches

There are several approaches to asset detection and registration. In this subsection of the thesis, some categories of these approaches will be enumerated and briefly explained, with an accompanying assessment of advantages as well as drawbacks to each method.

2.3.1 Manual

One method of keeping an overview of assets is the classical method of manual asset inventory. Organizations may decide to use a manual method of keeping inventory of hardware and software. Usually, such a method will involve a registration of hardware as it is delivered to the end user, changes to registration as changes are done by staff, and possibly executing a scheduled inventory at intervals to ensure that the inventory remains accurate.

The method works for many smaller organizations, where it is easier and cheaper to keep such an inventory rather than invest the time and resources in other methods.

Manual asset inventory has a low initial overhead compared to setting up dedicated server or software packages, and can usually be done in systems such as Microsoft Excel. More advanced examples of manual registrations can be found in the manual component of the popular helpdesk and

network inventory software Spiceworks, as seen in figure 2.2

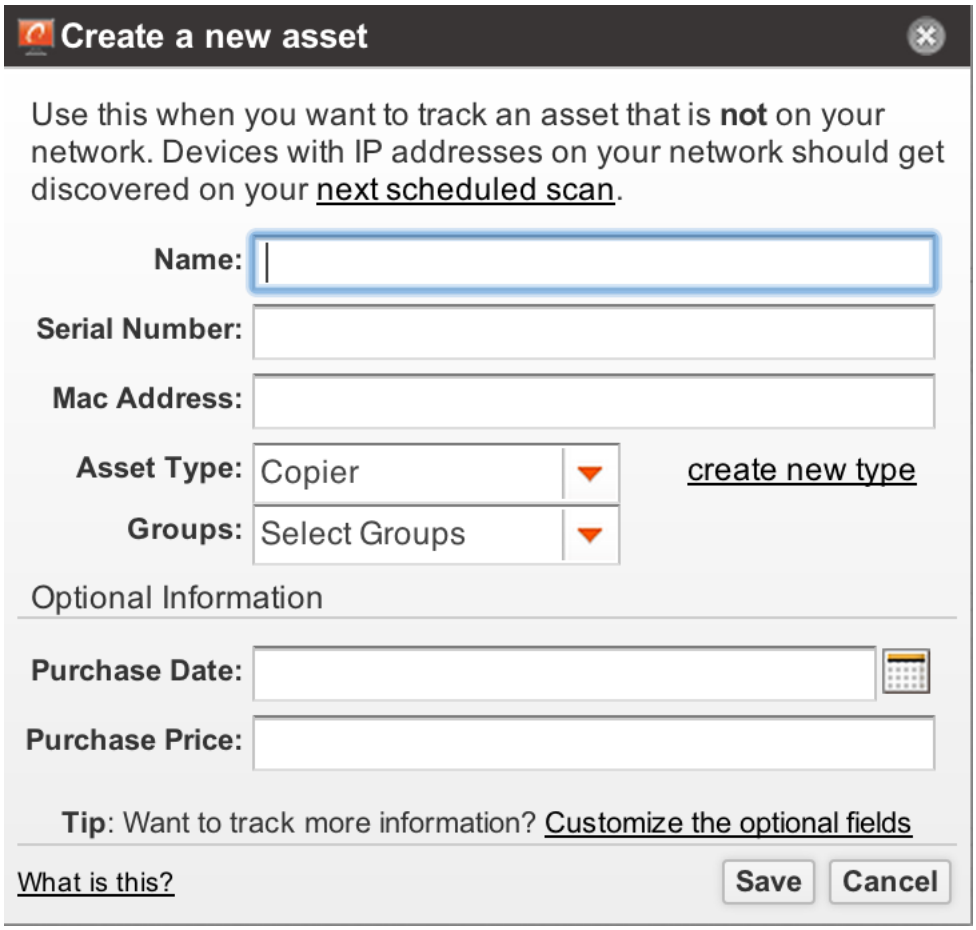


Figure 2.2: Spiceworks manual asset management example

There is also a lower overhead in skill requirements when compared to the other methods. Someone registering such information will likely not have to have much specialized skill to detecting and registering hardware and software on computers.

Given that the analysis of the hardware and software is manual, it is also possible to access a lot more data when compared to other methods mentioned in this thesis. It does not face the same constraints as passive network based asset detection or scanning solutions, which can only do analysis on traffic that travels over the network.

This method also offloads a lot of the computational overhead present in other methods to human-intensive workloads instead, which may be

wanted in situations where there is no equipment or budget for the work, but manpower is available.

On the negative side, this method is the least scalable method of those discussed in this thesis. While it usually has a low initial overhead, the ongoing overhead is much higher per asset inventoried. In many organizations, especially those bigger than a small office, the overhead cost per asset will quickly make it more economical to decide on a more scalable system of inventory.

2.3.2 Agent-based

A computer-powered analogue to the manual method mentioned above is to replace the manual process of having a human registering assets and keeping them updated with a software agent that does the same thing. There are two types of these agents. One is a software agent that will run on each machine being inventoried, registering software, configuration and hardware information. Alternatively, software will run on a central machine, which will log in and query the machine for assets using built in utilities. Spiceworks is one program which has the capability of logging into and inventorying machines on your network, as seen in figure 2.3.

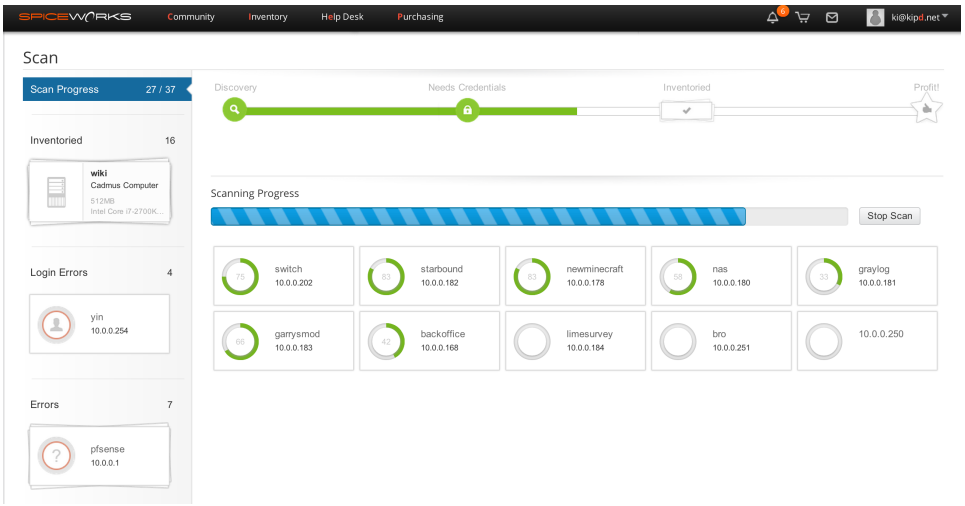


Figure 2.3: Spiceworks automatic asset management example

The method is very scalable, and can be done at a relatively quick pace depending on the implementation. This allows for continuous updates as

changes happen.

Information-wise, you are only limited to what the agent is capable of gathering, so depending on budget for development it is potentially possible to aquire all information available to the machine it is running on.

Some limitations to this approach must be mentioned.

It is a more intrusive method, when compared to passive and active network-based detection, since it requires either the installation of an agent on each machine being assessed, or access to these machines from a central machine doing remote calls to retrieve the same information. This leads to the following problems:

If the system is using an agent, the agent will have to be programmed so that it can run on all operating systems that should be assessed. Additionally, there might be differences in how each version of an OS will expose information that is relevant to the asset detection agent. As an example, Debian and RedHat Linux have different package managers used for installing software, and an agent will have to be aware of these differences should it be designed to access software installation and version information using their respective package managers.

Depending on the circumstance surrounding the use of the asset detection system, there are some problems with the intrusiveness of this solution. In many environments, it is not acceptable to all stakeholders to provide those in charge of the asset detection system access to all machines. This is especially true in cases where an external party is hired as a Managed Security Service Provider. In these cases, the external party may not be given complete access to the machines being protected. The same limitation exists in mixed networks, such as student networks at schools, where security has to be provided, but direct access is not available.

2.3.3 Active scanning

The first completely network-based asset detection method that will be covered in this thesis is the use of active scanning using tools such as NMAP. These tools use active network connections in order to gather interesting information about the machines on the network.

NMAP, as one of the archetypical examples of this type of scanners, has many features interesting to asset detection, including detecting hosts present in the network through several different forms of pinging, open port detection, operating system detection by fingerprinting the TCP/IP stack and software version probing for some protocols[12].

NMAP has reached its status as an invaluable tool to network administrators, security auditor and others due to its ease of use. Additionally, it allows for relatively quick scans of wide IP ranges, due to NMAP being designed for performance - using algorithms and parallel processing[14]. Such scans are usually not only for existing hosts, but also for listening ports – depending on launch options, of course.

One major advantage that active scanning tools have over passive network-based solutions is the possibility of detecting less active network-connected assets. That is, unlike passive asset detection systems it can detect hosts where there is no incoming or outgoing traffic that will reveal it.

Active scanning generally has a low overhead human-workload wise, since it usually only has to be installed on hosts installed at points in the network where the scans should be taken from. After installation, the scans can be initiated and left alone while they execute.

In the same vein, another advantage is that execution of such scans is usually relatively quick.

There are, however, some weaknesses to the active scanning approach. The limitations that are important to us both stem from the same problem: active scanning induces stimulus to get a response – which is then analyzed.

This means that a scan for open ports and services is limited to ports that respond to stimulus during the scan. A system like this will not detect transient services, client-mode software, nor services that are only available after other stimulus (like port knocking).

This also means that an active scanner is limited to addressable space for the network node from which it runs. It is unable to detect software or

other assets which are made unavailable behind a proxy or a NAT router. This kind of network setup is common where policy may dictate a division of networks into different zones. This problem could potentially be bypassed by distributing more scanning nodes into each of the networks where scans are required.

A final disadvantage to active scanning with tools such as NMAP is that they are relatively noisy when used in environments with IDS, generating alerts in many IDS tools.

2.3.4 Passive TPC/IP fingerprinting

A passive approach to asset detection is the use of passive TCP/IP fingerprinting. This concept bases itself on the same features as those used in several active scanning tools such as NMAP, and is based on detecting default settings in the IP stack corresponding to fingerprints for different operating systems. It separates itself from active scanning tools by not generating stimulus, only working passively by reading traffic generated during normal operation. Different operating systems will often have a measurable difference in settings – which will generate responses with specific features. These setting differences are usually recognizable in the TCP headers, but also on the IP headers.

p0f, a tool which pioneered many of these techniques, uses the initial client-originating SYN packet, as well as the server-originating SYN+ACK packet, to attempt to fingerprint the operating system of both network nodes. Features, such as TCP and IP header default settings, ordering of TCP options and other quirks are used to detect the operating systems.

This method of asset detection is scalable, since it works passively. Additionally, it does not need to do deep packet inspection, since it in the case of p0f only needs to look at the initial packets of the TCP handshake for information. This lowers the memory overhead in the machine doing the analysis of the network traffic.

The negative to this technique is that it is restricted to detecting the OS TCP/IP stack only, which offers a very limited amount of information when compared to other techniques presented in this chapter.

2.3.5 Software user-agents

Another approach, which can be used both actively and passively is using the software server or user-agent strings for software detection. Server and User agent strings are strings sent as part of the protocol to identify the version of the client or server that is communicating.

This type of fingerprinting is present in both NMAP and p0f, achieving the goal of detecting specific software and their version. In the p0f case, the information is used to indicate which OS is present on the node, by mapping software by its OS availability. An example NMAP scan for software versions present on a host can be seen in listing 2.4

```
Nmap scan report for 10.0.0.1
Host is up (0.0030s latency).
Not shown: 996 filtered ports
PORT      STATE SERVICE     VERSION
53/tcp    open  domain      dnsmasq 2.68
80/tcp    open  http        lighttpd 1.4.35
443/tcp   open  ssl/http    lighttpd 1.4.35
5666/tcp  open  tcpwrapped
```

Figure 2.4: Example nmap scan for software versions

The advantage of this approach to detecting assets is that it is relatively accurate, relying on payload data to identify assets. It can also be done passively, which lowers the network overhead for using it to detect assets.

The disadvantage is the resource usage as compared to TCP/IP fingerprinting. Since this kind of traffic analysis works on data from the payload of network traffic, the system will have an overhead for analysis. A passive system will have to do reassembly of packets until it finds the strings it is looking for, and an active scanner will need to establish the connection and exchange data to get the strings.

2.3.6 Flow detection

A final approach to passive asset detection was presented in UiO Master's student Mats Klepsland's thesis "Passive Asset Detection using NetFlow"[10]. The technique relies on using NetFlow flows to detect network

assets, communicating ports, and OS update servers.

NetFlow is a format for describing communication flows inside a network, summarizing statistics for entire sessions in single entries. These statistics can be used to map communicating hosts and ports in a network. The data can be generated directly on Cisco routers or from pcap data.

An advantage of this approach is that it is very scalable, given that data is given in summarized form from the sensors themselves. Additionally, the summarized data does not take up much space, and therefore historical data can be saved for later analysis, even against new fingerprints.

The main problem with this technique is that we only have some very basic information about the network traffic, which makes it hard to accurately detect deeper information about the hosts. Detecting service types running on a host by their port number is possible, but it is impossible to accurately know what specific service is running on the port.

2.3.7 The case for passive IDS-based asset detection

A final type of approach, building on some of the above described passive techniques, is passive IDS-based asset detection which is presented in this thesis. There are some points in favor of designing an asset detection system around the concept of intrusion detection systems.

Less intrusive

First and foremost the use of an IDS for the purpose of categorizing assets is in many ways less intrusive than many of the solutions above.

A passive network solution to asset detection requires one or more sensors placed centrally in the network, able to listen to passing traffic.

Since the solution is passive, it does not generate any network traffic, and it leaves no risk of harmful side effects of probing activity. Since all processing is done on the sensors/outside of the network, the use of such a system minimizes the resource overhead on machines and on the network of the audited organization.

Unlike the agent-based solutions above, it does not require direct access and installation/execution privileges on the assets being audited. This makes the solution ideal for Managed Security Service Providers, as they will typically require this information for informed decision-making on security-events, but often do not have access to central asset information, nor direct access to network clients.

Scalable

The solution is also very scalable. Initial overhead in terms of manpower is small, since the setup requires only the installation of a single or a few sensors depending on network throughput and design. When compared to installing agents or manually indexing all machines in the network, this is a very big reduction in work.

For the same reason, the deployment of new signatures is reduced to just modifying the sensors, rather than all machines in the network.

A final consideration is that this analysis can also be done asynchronously, in that we can collect data over time, and later analyze it. This possibility allows for more analysis to be done at off-peak hours, at the cost of data storage for pcap files.

2.4 IDS-based asset detection

What separates the system described in this thesis from the previously mentioned passive solutions is the choice to leverage the strength of an IDS to gather and analyze data for the asset detection system. This brings some additional capabilities to the table, and removes some of the inadequacies in the abovementioned systems.

The most obvious result of moving to a system where we can do analysis over payload data is that we have more data to analyze, which allows us to form more accurate signatures. It also allows us to make signatures for cases where under other solutions it is impossible to distinguish between different types of assets. In the case of NetFlow analysis, a source of false

positives for OS detection was theorized to be update server reuse for Microsoft products, where Microsoft Office updates looked like Windows updates to the system.

As a result of the increase in available data to analyze, there is also a sharp increase in the amount of "low hanging fruit" in terms of information available for analysis. Many of the signatures introduced in the Implementation chapter are based upon relatively simple analysis of URLs requested by software.

2.5 Threats against asset detection

Throughout this thesis, it is important to keep in mind that it is very possible, and in many cases simple, to subvert the techniques used to detect and identify assets. As far as is possible, the thesis describes how such subversion can be done in the specific cases in the Implementation chapter, but this section describes in broad stokes the three primary methods that are relevant to us.

2.5.1 IP stack scrubbing

IP stack scrubbing is the concept of modifying the default settings of the TCP/IP stack or by modifying the packets after they have left the stack . The end result is the same: to a passive or active listener using TCP/IP fingerprinting, the packets can be made to look originate from an entirely different OS than from what it actually is.

The software to modify the characteristic TCP and IP header output is available for many platforms. OSfuscate is a tool for Windows, which modifies default settings used by the network stack in windows. This is done by modification of HKEY values in the registry of Windows. The tool contains profiles of default settings for several system types, as can be seen in figure 2.5.

IPPersonality takes a more limited version of this concept to defend linux hosts against NMAP type OS detection. It tries to detect abnormal traffic of the type NMAP generates, and responds in place of the system

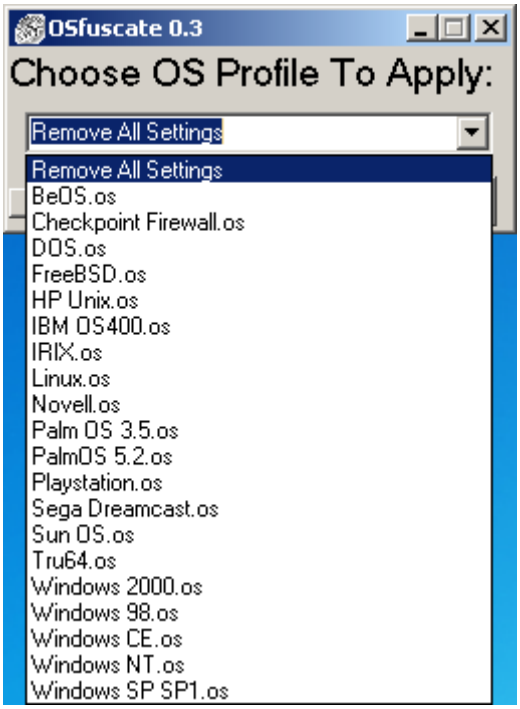


Figure 2.5: OSFuscate OS profile selector

with packets corresponding to a user-chosen OS profile.

2.5.2 User agent/Server string modification

Another modification which is more commonly use on the Internet is the modification of the headers for User agent strings and Server strings in HTTP communications.

User agent strings have been used for compatibility purposes in web development for a long time, and have lately been used for separating between mobile and desktop users.

The modification of this string is usually not detrimental to the end user, and in some cases this kind of modification may be necessary to let the end user gain access to services which are limited to only certain user agent strings. This is the case for the popular video streaming service Netflix, which does not allow most Linux users access to its services. In conjunction with the plugin system pipelight, user agent switching allows end users access to artificially constrained platforms like Netflix[2]. For this reason,

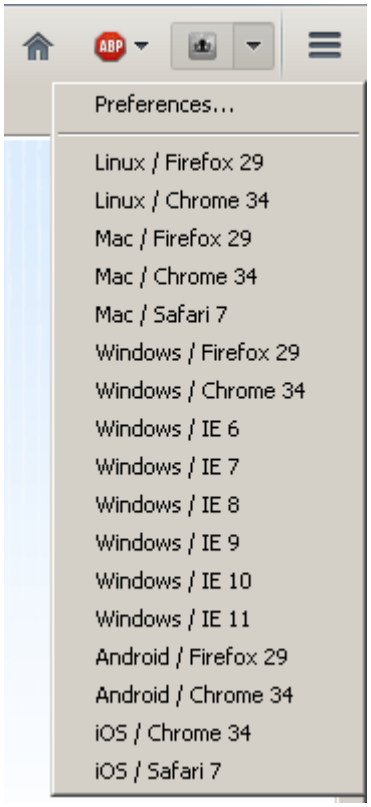


Figure 2.6: User agent switcher for Firefox

many of the major browsers have plugins that allows for rapid changing between different User agent strings, as can be seen for Firefox in figure 2.6.

The Server string has been even less useful for the most part, and many encourage administrators to modify settings to ensure software version data is not leaking to an attacker. In the case of the popular Apache web server, the capability of entirely modifying the server string is provided by the mod_security module project.

2.5.3 Behavior modification

A more active type of fingerprinting countermeasure is the active use of behavior that the end user knows will mislead the asset detection system.

This could include connecting to update servers for software that is not installed, opening listening ports and in general emulating the behavior that asset detection systems are trying to detect.

This thesis does not discuss methods to avoid fingerprinting in detail. The reason for this is twofold.

A sufficiently determined adversary can generate as much traffic as she likes to confuse us (as in figure 2.7), and there is very little we can do about such behavior without direct control over the network connected machine used to generate the misleading data.

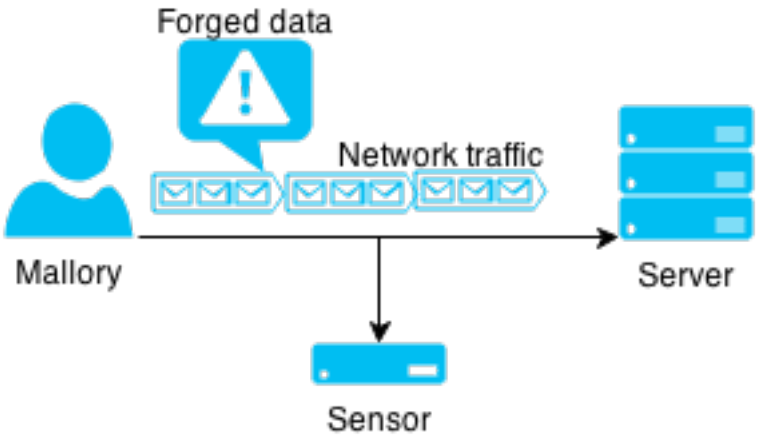


Figure 2.7: Malicious user injecting false traffic

Secondly, the proposed use of this system is an asset detection system for an organization or a service provider, where it is assumed that the assets being protected are not controlled by our adversaries, but rather by end users which wish for optimal protection.

Since we in most cases cannot eliminate threats against fingerprinting, an option to reduce the risks of manipulated data is to be aware that they exist, and to intelligently assess the data the system has gathered. One must be very aware of inconsistent data, and seek information gathered in an alternate manner if the data is suspect.

In the implementation of this thesis, this was partially achieved by not aggregating together less trustworthy sources of data, and thus allowing the analyst to make a decision on the how sufficient the data is. This was done on all entries gathered by hostname signatures, as these vary in how trustworthy they are due to the source of the data (HTTP hostnames and

DNS replies). While a web server can modify what hostnames it responds to, and a malicious user can inject requests for hostnames it wants to imply are hosted on the web server, the malicious user will have to gain control over the DNS server or DHCP server to inject data confirming the maliciously injected data for these alternative sources of hostnames.

2.6 Software lifecycle

A final concept that needs introduction before use later in the thesis is the Software lifecycle. This lifecycle describes the general common stages in the "life" of software while in the hands of the owner and end-user, and can be seen in figure 2.8. The concept is mentioned here because the Implementation chapter is divided into these stages, since each stage usually is shared across many different types of software.

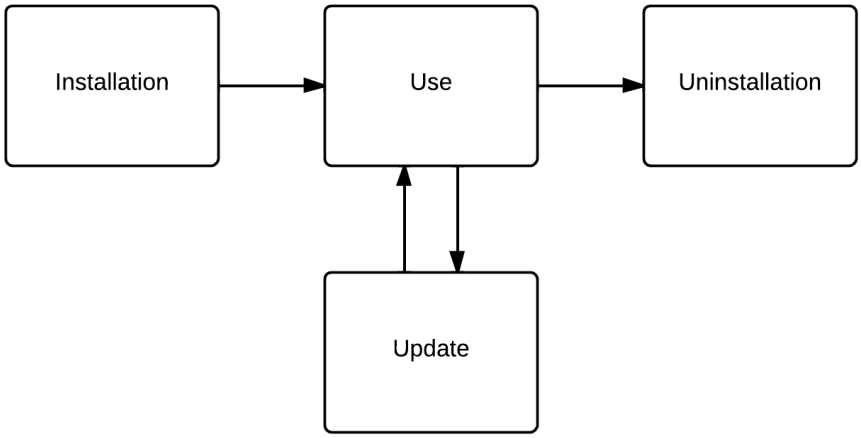


Figure 2.8: Simple state diagram for the software lifecycle

Most software will go through all of these stages during its lifetime on a machine. Each stage can contain multiple indicators that can be detected on the network.

The Installation stage can typically contain requests for installation files, requests for configurations or updated default data. Optionally, and less to the support of the software installation, is the use of registration or checkin traffic for statistics purposes for the developer.

The Use stage is the stage which is most varied of the stages. The fingerprints present in this stage is entirely dependent on the purpose of the software in question. Typical indicators are ports being opened, specific telltales in the protocols being used by the software or traffic towards specific IP addresses or hostnames.

The update stage generally consists of either checks for updates, or the download of said updates when they are available. Today, most updates are handled over HTTP, which makes much of the analysis very simple.

The final stage is the uninstallation stage. In an asset detection system, this stage can only be used to unregister or delete asset information.

Chapter 3

Design

The goal of this thesis is to develop a method for passive asset detection on higher levels of the OSI model based on the use of intrusion detection systems.

This chapter is dedicated to explaining the research methodology used for the research in this thesis. This is followed by a section discussing some of the relevant types of information an asset detection system may be expected to gather, the progress of gathering and reducing the data necessary for making fingerprints, and some of the tools I found useful to develop said signature-scripts for detecting assets. The tools are divided into the following 3 categories:

- Gathering network traffic
- Narrowing the traffic to the traffic of specific applications of interest
- Analyzing and extracting indicators

3.1 Research method

My research project has been structured to follow the research methods of a Design Research Process, as presented by Vaishnavi and Kuechler[16].

This method is divided into multiple stages of development. At the *awareness of problem* stage, the research problems were identified. Following this, in the *suggestion* stage, tentative designs for solutions to the research problems were developed. In the *development* stage, a potential solution to the research questions was developed and implemented. The next stage,

evaluation, was the stage in which the solution was evaluated for performance. In both of these stages, when encountering situations that did not work according to the proposed solution, the project followed the circumscription arrow seen in 3.1, taking into account the knowledge of what did and did not work on to the next iteration of developing the solution. The last stage seen in the diagram, the *conclusion*, was the stage in which the final accepted results were reached.

This research method follows many of the same patterns as iterative forms of software development. Since this type of development is something I have experience with, and since a large part of the project was to develop an asset detection system, using a research method which complements this seemed to be a prudent choice. Additionally, the iterative nature of finding and refining and redeveloping fingerprints for assets allowed me to carry over the methods to the smaller parts of the research project.

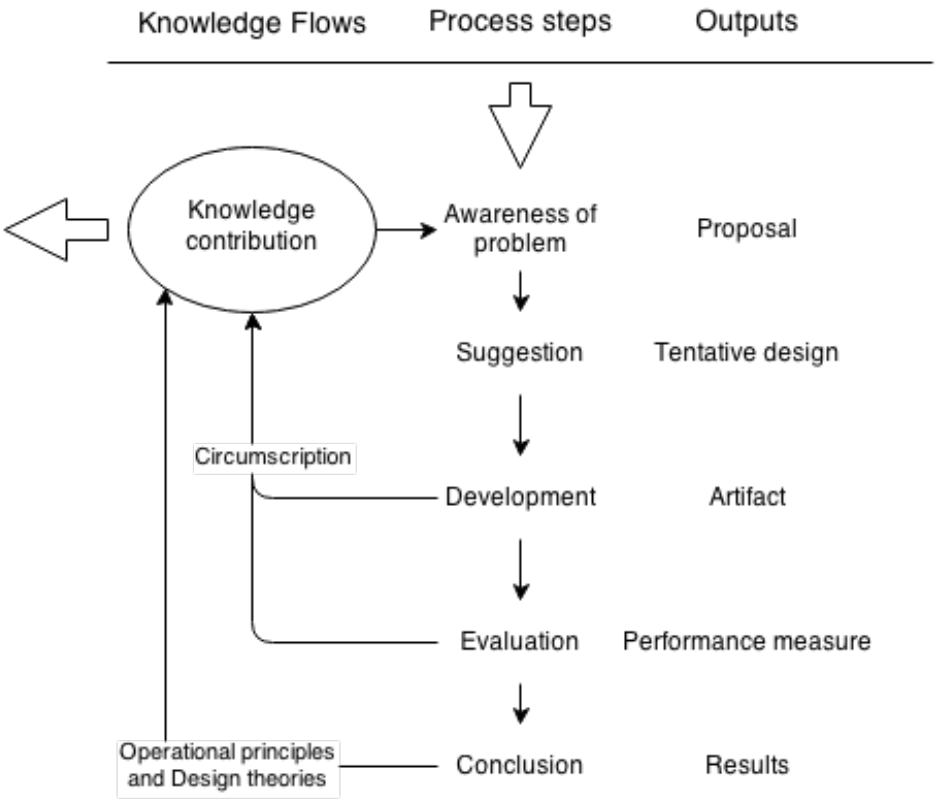


Figure 3.1: Design Science Research Process Model

3.2 Fingerprinting overview

One of the primary things to remember when making asset detection fingerprints is the Unique pattern mentality[15]. This means that we have to isolate behavior or indicators which are specific to certain software or other assets.

To begin with, it is important to understand what data is relevant for an asset detection system. Previously in this thesis, arguments were made for motivations behind using asset detection. The data we wish to find per network host reflect these motivations.

The data which is usually accessible and interesting to a software asset detection system are the following:

- Operating system, version and platform arch
- Software – and its version
- Ports with a listening service

The Operating System of the machine is relevant for several reasons. First and foremost, it is the primary software of a machine, and governs most of the internal functions of the machine it runs on. Operating systems, as the provider of software's environment, is also often a limiting factor in what software will be available on the machine in question. When a machine with a Linux based operating system is the only system on a host, it is possible to disregard entire classes of attacks – such as IIS specific vulnerabilities. Since IIS is a proprietary web server only made for Windows operating systems, any such attack is likely to be irrelevant to the host in question.

The instruction set architecture of a processor defines the low level codes and registers available to the software running on it. This means that software and operating systems will have to be compiled specifically to the architecture of the processor. This is usually called the arch of an OS. The arch of a system is not quite as interesting by itself, as most software is not arch-specific unless it uses a lot of low-level code. It is, however, interesting for distinguishing different operating systems installations behind a network address. If there are mutually exclusive indicators, such as different OS arches registered to the same IP address, the information can be

used to pinpoint IP addresses that are assigned to a NAT or proxying host.

Knowing what specific software - and its versions – runs on a machine is primarily interesting because of the vulnerabilities they represent. For each software installation and each of their versions, there will in most cases be vulnerabilities that apply specifically to them.

Detecting open ports on a host is interesting because it will give an indication of the class of software that resides on a network node, if not the software itself. An open TCP port 80 on a host will generally be a good indication that a webserver runs on the host. Some software also use specific ports by default, and when these ports are not used by other software, they can be used to infer that said software will be running on the host. An example of this is the game Doom, which will generally listen on port 666 (tcp and udp).

Additionally, as mentioned in the motivations section, the OS and software versions are also relevant for standardization of the software stack, and for keeping an overview over available assets.

In the information security field, there is one other activity that closely mirrors what we experience when gathering signatures for detecting software as assets. Gathering IOCs, or Indicators of Compromise, is a well-known task that is usually associated with making signatures for detecting malware. With the exception that malware is considered to be malicious – and therefore must be analyzed in a safe environment - there is little difference in the techniques which can be used to gather fingerprints which are unique to the software.

IOCs as a field is more far-reaching compared to the information that someone writing signatures for a passive asset detection system will need. Since gathering IOCs also extends to artifacts on the machine running the malware, tools and techniques will also be looking for indicators like memory artifacts, md5 sums of files, computer libraries loaded, as well as other indicators. Given that the only place where a passive network-based system has access to data is on the network layer, we generally only care about IOCs which are possible to track in the network.

When analyzing software, it is important to understand that while some

software sends network traffic of its own initiative, other behavior needs to be induced. This can be characterized as stimulus/response – and is important when writing signatures to detect behavior that we are interested in. Some software allows the user to induce a check for updates online. The stimulus in this case, is the user clicking "Check for updates online". The response – from the network perspective – is a request for the software's update server. Inducing stimulus to get the desired response allows for a faster workflow when collecting indicators for software installations, and can be completely necessary to gather fingerprints that are not generated by automatically scheduled behavior.

3.3 Procedure and tools

This section describes the general procedures used to gather the data necessary to make asset fingerprints. Each subsection contains some choices in tools that may be helpful in the process.

3.3.1 Gathering network traffic

When gathering network traffic, we have a rich set of options in tools to use. During my work, I limited myself to two of them – tcpdump and Wireshark. Many of the tools available for analysis of network traffic are based around the libpcap/winpcap libraries. Most of the tools I used for further analysis while working on this thesis are based around these libraries.

Tcpdump is a tool for capturing and analyzing packets. It is available on most unix-like operating systems, and is therefore a good tool when investigating the network traffic of software running on hosts with these operating systems. Additionally, I used tcpdump (as seen in listing 3.2) to capture data on a dedicated sensor based on Linux, with port-mirroring from a central switch in my own test environment.

```
sudo tcpdump -i eth0 -w filename.pcap
```

Figure 3.2: Example tcpdump command

Using tcpdump to record all traffic allows for some important parts of

the workflow which is useful, or even necessary, for writing good asset detection scripts.

One of the primary reasons for capturing full packet dumps of the relevant traffic is that after capture you can run the recorded file through several different tools, as well as testing your finished asset detection scripts. Keeping the full dump allows for reproducibility of the work, and insures you can check improved scripts for unintended loss of functionality.

On windows hosts, one can use the tcpdump alternative windump. Alternatively, if working in a graphical environment, tools such as Wireshark allows for packet captures, as well as further analysis of the captured packets.

3.3.2 Narrowing the traffic to the traffic of specific applications of interest

Once we have recorded a network capture containing the indicators we are after, we have to separate out said indicators from other behavior.

There are two good ways of separating out the indicators relevant to the program we are investigating; either by IP or by port.

Separating out indicators by IP is useful in the case where a dedicated sensor is capturing the data (along with other network traffic). The host(s) behind the IP in question should ideally reduce the amount of nonrelated network communication while the software generates its traffic. A possible way of implementing this is to use a dedicated virtual machine to have a clean and reproducible environment for the investigated software to run on.

Virtual machines are also useful in that you can make a custom network with several virtual machines to observe behavior in isolation.

Separating indicators by ports is useful when investigating software running on a workstation where other software may also be communicating, but where we have access to execute commands. By detecting which ports are used by the software, we can separate out traffic that is related to the software in question. This allows for a reduction in overhead time

cost when compared to setting up a dedicated environment to investigate single pieces of software.

One tool, which can be helpful in cases where one wishes to collect fingerprints, is Mandiant Redline. This software suite was made for gathering and analyzing malware Indicators of Compromise. It consists of a program that allows generating agents that can be run on computers suspected of containing malware, which in turn gather various types of information which can be used to make IOCs.

For our purposes, the tool is used in a 3 part process. Generating a Collector, running it on a machine with the software we wish to gather data on, and analyzing the data in Redline.

Generating the Collector is a relatively simple task. The important part of this is to make sure that the collector will be gathering port usage and DNS lookups.

Following this, the collector is run on the machine where the analyzed software resides. Here it gathers various indicators as they occur, and record them to disk.

Finally, the report can be generated by Redline using said data to present the indicators gathered, as seen in figure 3.3.

Alternatives to this process is the use of TCPView for windows, which enumerates all TCP and UDP endpoints on a windows-based computer, along with the process which owns the port. The utility is available from microsoft's webpages. The use of this tool to find ports in use can be seen in figure 3.4

On the GNU/Linux, *BSD and OS X side, one can use the command line utility `lsof` with the `-i` option. `lsof` is an utility to display all open files and the programs which open them. The `-i` option limits the output to Internet files (open ports and connections). The utility is present on many unix-like systems. An example run can be seen in figure 3.5

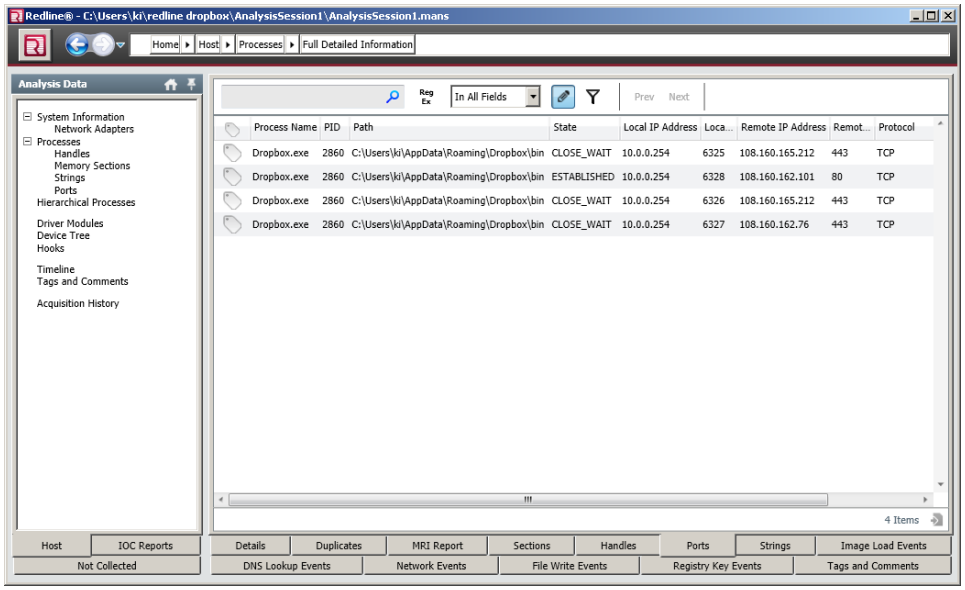


Figure 3.3: Port use identification with mandiant redline

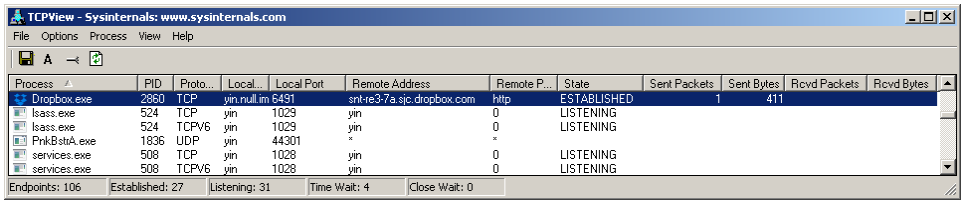


Figure 3.4: Use of tcpview to identify port usage

3.3.3 Analyzing and extracting indicators

Finally, when we have an initial starting point to go over the network packet dump, we can investigate the content of the dump for indicators.

The primary tool for me was the use of Wireshark, which allows for sorting packets by expressions, and interpretation of some protocols. Another very useful tool when investigating is the built in facilities of Bro. By activating the subset of scripts relevant to your analysis – like the HTTP logs, and then investigating for indicators relevant to the software being analyzed one can make a script using the built in scripting facilities, and iteratively work towards making a script which outputs the desired asset indicators.

```
kisume:~ ki$ lsof -i | grep Dropbox
Dropbox     695   ki   20u  IPv4 0x7f0902d28f8f5337      0t0  TCP kisume.null.im:53839->snt-re4-6d.sjc.dropbox.com:http (ESTABLISHED)
Dropbox     695   ki   25u  IPv4 0x7f0902d2806f705f      0t0  UDP *:17500
Dropbox     695   ki   28u  IPv4 0x7f0902d282617b4f      0t0  TCP *:17500 (LISTEN)
Dropbox     695   ki   37u  IPv4 0x7f0902d28261e337      0t0  TCP localhost:26164 (LISTEN)
```

Figure 3.5: Example use of the lsof command

3.4 Bro

Bro is a Network Intrusion Detection System (NIDS) with a clear separation of the engine and policy[13].

Architecturally, the Bro IDS is divided into 3 parts: the libpcap based network sniffer for passive traffic gathering, the main engine, and the policy scripts. This can be seen in figure 3.6

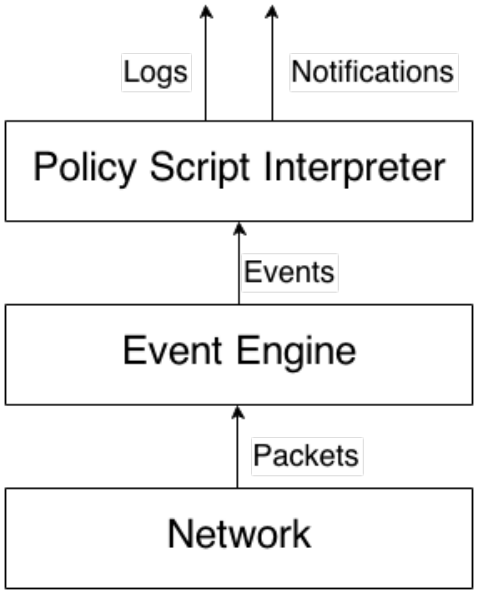


Figure 3.6: Bro architecture

Traffic is passively gathered from an interface on the sensor, usually off a mirrored port from a central part of the network to be protected by the sensor. Packets are passed directly to the event engine.

The event engine assembles the various packets into bi-directional streams of data. These streams of data are then analyzed by the various protocol analyzers in the event engine. These analyzers then trigger events based on the traffic it detects. These events are then passed onto the policy

scripts that reside in “scriptland”.

Event handlers in the policy scripts are then invoked with the data from the event engines. These scripts containing event handlers constitute the policy of the network sensor. There are a number of scripts included with the Bro distributions, but administrators are expected to add more scripts to fit their own needs as well.

This separation of analysis of protocol and analysis of content allows for a more effective working process when making IDS signatures – or in our case, fingerprints of assets.

As an example, a client `10.0.0.2` makes an HTTP request to a webserver `10.0.0.3`, requesting the root document for a hostname example.com. On the wire, the traffic might look something like that seen in figure 3.7

```
GET / HTTP/1.1
Host: example.com
```

```
HTTP/1.1 200 OK
Date: Mon, 16 Jun 2014 17:16:54 GMT
Server: Werkzeug/0.8.1 Python/2.7.3
Content-Type: text/html; charset=utf-8
Content-Length: 13864
Vary: Accept-Encoding

<!DOCTYPE html>
[traffic truncated]
```

Figure 3.7: Example HTTP request and response

To map the relationship between the Host field of an HTTP request and the responding server banner (`Werkzeug`), a manual process will have to divide the traffic into request and responses, a parser would have to find the Host field in the request, and then another parser will have to find the Server field in the response. In a bro script much of this work is done already, and an analyst can write a script like that seen in figure 3.8

This script will output the text seen in figure 3.9 for each time the example request above is done.

```
1  event http_all_headers(c: connection, is_orig: bool, hlist:
       mime_header_list)
2  {
3     if ( !is_orig )
4        for ( i in hlist )
5        {
6           if (hlist[i]$name == "SERVER")
7           {
8              local software = hlist[i]$value;
9              print fmt("%s on %s is running %s", c$http?$host,
                   c$id$resp_h, software);
10          }
11       }
12 }
```

Figure 3.8: Example Bro script

```
1  example.com on 10.0.0.3 is running Werkzeug/0.8.1 Python/2.7.3
```

Figure 3.9: Example Bro script output

While the language utilized by Bro is domain-specific, it is a relatively powerful in a network context, and features a good library of built-in functions.

Bro also comes with a good number of scripts built in, allowing for rapid deployment of many frequently wanted functions for an IDS.

Bro is configured with an initial script which includes other packages and scripts, which then provide the functionality required to fulfill the policy. By default this script is the `local.bro` script (located in `$PREFIX/share/bro/site/local.bro`.

By including other scripts, such as `base/protocols/http`, the Bro runtime functions are extended to do things like logging all HTTP traffic to `http.log`. This is especially helpful, since HTTP request and connection logging can provide much of the data required for writing good scripts for detecting assets.

As an alternative to using scripts to analyze payloads, Bro also contains a signature framework, much like the rule engines found in Snort and other intrusion detection systems. This allows for efficient analysis of payload data, which can then be escalated to the scripts by catching the

signature_match() event.

Bro also comes with a program for parsing its own log files in the terminal. This program, bro-cut, allows for parsing Bro's formatted log files by field names, outputting line by line in a format specified at the command line, as can be seen in figure 3.10.

```
1 ki@bro:/opt/bro2/logs/current$ cat http.log | bro-cut "id.
    orig_h        id.orig_p        id.resp_h        id.resp_p"
2 10.0.0.16       51491    108.160.162.98   80
3 10.0.0.16       51447    10.0.0.181       80
4 10.0.0.16       51447    10.0.0.181       80
5 10.0.0.16       51447    10.0.0.181       80
```

Figure 3.10: Example use of brocut

This allows for further analysis and lookups in historical log data, letting us understand trends and seeing changes in requests over time. Since it reads and outputs over standard input/output in the terminal, this allows us to use standard unix tools like `grep`, `cut` and `awk` to further analyze traffic.

When analyzing network traffic, using bro scripts and logs becomes especially valuable. Much of the required information for writing fingerprinting scripts will be available in bro logs. This allows for analyzing historical data to write accurate fingerprinting scripts. Likewise, actively using the Bro engine through scripts while running and stimulating software allows for analyzing network traffic live. Alternatively, Bro can easily be invoked reading from a pcap file and custom scripts, allowing for easy analysis and iterative development of signatures using captured traffic.

3.4.1 Optimizing for speed

A consideration which is important when designing a system which will analyze realtime data is the question of speed. In a system where the analysis takes too long, the software will have to buffer traffic data for analysis when possible. If the buffers are filled, the sensor will start dropping packets to cope. This results in a loss of data which may be important to analysis, and gathering of asset data.

Beyond not doing more analysis than is necessary in scripts, and reducing operations which may be computationally expensive, there are two techniques which are used when writing the scripts used in the proof of concept made for this thesis. Both amount to the same: reduce the incoming events, and abort analysis as soon as we know the event is irrelevant for the fingerprinting script.

Using the variable `is_orig`, which is present in many of the events generated by the analyzers of Bro, one can determine the direction of the data in the event when compared to the full connection flow. `is_orig` is true when the event is generated on data from the originating host in the connection - in other words the client. This allows us to abort early on events where only traffic from either the server or the client in a connection is relevant, and the event was triggered on the opposite side of the connection.

Another technique, more useful specifically for asset detection, is defining an asset detection subnet variable and checking if the subject of the event captured by the signature is not in this network. This way we can avoid running analysis of the event in cases where the detected asset data is outside the investigated subnet, since the asset data should not be gathered for network nodes in these cases.

Chapter 4

Implementation

This chapter of the thesis describe several methods and situations where it is possible to gather data on assets in an organization. The chapter is divided into several sections according to a stage in the previously outlined concept of the software lifecycle. Each stage in the lifecycle contains a few different ways of detecting some common indicators that are common to that stage. Following these sections is a section on detecting the hostname of clients. Finally, there is a section devoted to detecting proxy and NAT addresses, which is a special case in that it does not apply to a single asset but rather a network of them.

4.1 Installation

This section of the chapter describes two categories of fingerprints that occur during the installation phase of software.

4.1.1 Package managers

The concept of package managers is a prevalent concept among Linux and BSD based operating systems. They are used to install most software from central repositories maintained by the various groups and projects behind the individual distributions. Since the alternative is usually to manually configure and compile software packages, this method of installation is usually preferred.

During the installation of software using these package managers, various files are downloaded from the central repositories. In the examples

shown in figure 4.1, the requests for installation files are done over HTTP. This is the case for many package managers. This makes it very simple to make Bro scripts to detect the installation of software on servers utilizing these package managers.

```
1 #fields client   server  port    method  hostname        uri
2 10.0.0.155       137.226.34.42   80      GET     cdn.debian.net
      /debian/pool/main/h/htop/htop_0.8.3-1_amd64.deb
3 10.0.0.155       137.226.34.42   80      GET     cdn.debian.net
      /debian/pool/main/p/python3.1/python3.1-minimal_3.1.3-1
    _amd64.deb
4 10.0.0.155       137.226.34.42   80      GET     cdn.debian.net
      /debian/pool/main/p/python3.1/python3.1_3.1.3-1_amd64.deb
5 10.0.0.251       193.35.52.51    80      GET     no.archive.
    ubuntu.com   /ubuntu/pool/universe/h/htop/htop_1.0.2-2
    _amd64.deb
6 10.0.0.22        193.35.52.51    80      GET     no.archive.
    ubuntu.com   /ubuntu/pool/universe/m/mongodb/mongodb-dev_2
    .4.6-0ubuntu5_amd64.deb
7 10.0.0.22        193.35.52.51    80      GET     no.archive.
    ubuntu.com   /ubuntu/pool/universe/m/mongodb/mongodb-
    clients_2.4.6-0ubuntu5_amd64.deb
```

Figure 4.1: APT-GET HTTP requests

Ideally, this allows the asset detection system to get a complete overview over all software installed on a system. In practice, many operating systems will initially use files present on the installation media when installing the base system, but all later installations and updates will likely be detected.

While it is always possible to fake such update requests, it is unlikely that a real system will provide false positives. Since updates and installations are a necessity for most systems, such traffic will typically be correct.

In the thesis implementation of the detection script for the use of the package manager apt-get (used by Debian derived distributions like Ubuntu and Mint) one can parse out additional information about the system. From the example above (which contains requests from both a Debian and a Ubuntu based system) we can gather the Linux distribution, specific software being installed and its version based on the requests for installation files.

4.1.2 Installation checkin traffic

Another possible fingerprint that occurs in some software is the concept of installation checkin. This is a behavior where software, upon being installed, will check in to the developer to notify them of installations – usually for statistics purposes.

In most cases this will yield a fingerprint of a specific software, but sometimes the installed software will also report additional information.

```
1 #field  client  server  port    method  hostname        uri
2 10.0.0.254      195.18.221.152  80      GET     subscriptions.
    amd.com   /driverinstalled/index.html?VID1=1002&DID1=6798&
    PID1=AMD Radeon Graphics Processor&SSVID1=1002&SSID1=0b00
```

Figure 4.2: AMD installation checkin

In the example in figure 4.2, gathered during the installation of AMD Graphics Drivers 14.06, the request was done after installation was finished. This request does not only indicate that an AMD graphics driver has been installed, but also what graphics card hardware is present in the machine doing the request.

The limitation to this type of fingerprint is, as with most others, that anyone can do the request in order to emulate the fingerprint. To the end user there is no downside to either emulating or suppressing this type of request, as it is only used for statistics for the vendor.

Most software that does exhibit this behavior has been seen doing this type of callback over HTTP, and are thus usually easily detected using a simple Bro script.

4.2 Use

This section discusses a series of asset fingerprints that occur during normal usage of software., and how this can be used for asset detection.

4.2.1 Greeting banners

A greeting banner is a part of the initial greeting for many protocols. The service banner is usually sent immediately after a client has connected to a service, and usually contains a string denoting the software version of the server.

Among the more notable protocols where the greeting banner is present are the SSH, FTP and SMTP protocols.

Generally, the banners will contain the software and its version, sometimes along with other information. In the example below, the banner used by `bro.example.com` shows that it is an Ubuntu-based operating system running on the server. At the same time, the `ssh.example.com` server is configured to only broadcast its OpenSSH version.

```
philipcs$ nc ssh.example.com 22
SSH-2.0-OpenSSH_6.1
```

```
philipcs$ nc bro.example.com 22
SSH-2.0-OpenSSH_6.2p2 Ubuntu-6
```

```
[philipcs@safir]~% nc smtp.uio.no 25
220 mail-mx2.uio.no ESMTP Exim 4.80 Tue, 24 Jun 2014
    19:30:52 +0200
```

Figure 4.3: Some example banners

There are some limitations to the use of banners for asset detection, though. While the banner is present by default in many standard configurations, for the major protocols the greeting is not essential to the protocol. Therefore many system administrators decide to change the host banner in order to not broadcast the software and version running on specific ports. In the case of the protocols mentioned above, most of them usually have a relatively easy way of disabling this greeting banner. OpenSSH does not offer the possibility to do this, as the string is part of the protocol . It can, as seen in figure 4.4, be limited to just the OpenSSH version, without the OS identifier, by changing the `DebianBanner` configuration option.

These banners are generally sent during the initial connection to the service, and are therefor possible to extract using the Bro signature framework.

```
philipcs$ nc bro.example.com 22
SSH-2.0-OpenSSH_6.2p2
```

Figure 4.4: Reduced banner information

Additionally, some of Bro's analyzers for protocols have built in software and version detection. Notable among these are the presence of scripts for detection of SSH software – which includes both server and client versions.

4.2.2 User agents and plugins

Among the more easily available fingerprints in both traffic amount and asset information amount is the use of user agents. Specifically HTTP clients are often available for analysis, given that so much of internet traffic runs over HTTP. While other types of software also use user agents (see OpenSSH above), HTTP clients seems to have the largest proportion of traffic where user agents are relevant. This section will therefore mostly deal with HTTP.

User agents usually identify the client software and version, usually so that the server can handle traffic according to the clients' capabilities. In the case of HTTP clients, serving web pages according to the client type was especially prevalent during the early web browser wars, and is still used for this as well as other purposes like serving mobile versions of webpages.

In the case of HTTP clients, the user agent will generally contain at least a software name and its version. Many clients, however, will also serve a wealth of other information along with this. The user agent information gathered by Bro's internal software framework can be seen in figure 4.5

Chrome, in the figure 4.5, will also inform the server of its operating system and architecture, while the Valve Steam client will also inform us of its language settings. Additionally, some clients will inform the server of the plugins available on the client, such as the .Net library in the MSIE case above.

While not a requirement for the HTTP protocol, specification of a user agent is generally seen in most requests. It will be found among all the other headers sent in each request coming from the client.

```
1 #fields host    software_type    name    version.major    version
    .minor    version.minor2    version.minor3    version.addl
    unparsed_version
2 10.0.0.16       HTTP::BROWSER    Chrome   18       0       1025
        166      -       Mozilla/5.0 (Macintosh; U; Macintosh;
    en-US; Valve Steam Client/1401381906; ) AppleWebKit/535.19
    (KHTML, like Gecko) Chrome/18.0.1025.166 Safari/535.19
3 10.0.0.16       HTTP::BROWSER    Chrome   35       0       1916
        153      -       Mozilla/5.0 (Macintosh; Intel Mac OS X
    10_9_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome
    /35.0.1916.153 Safari/537.36
4 10.0.0.16       HTTP::BROWSER    Chrome   35       0       1916
        153      -       Mozilla/5.0 (Macintosh; Intel Mac OS X
    10_9_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome
    /35.0.1916.153 Safari/537.36
5 10.0.0.16       HTTP::BROWSER    Crash%   20       -       -
            -       Reporter/1       Crash%20Reporter/1.0
    CFNetwork/673.2.1 Darwin/13.1.0 (x86_64) (MacBookPro11%2C3)
6 10.0.0.251      HTTP::BROWSER    Python-urllib    3       3
            -       -       -       Python-urllib/3.3
7 10.0.0.16       HTTP::BROWSER    iTerm    1       0       0
            20140518         Sparkle/313      iTerm/1.0.0.20140518
     Sparkle/313
8 10.0.0.168      HTTP::BROWSER    MSIE     8       0       -
            -       -       Mozilla/4.0 (compatible; MSIE 7.0;
    Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727; .NET CLR
    3.0.4506.2152; .NET CLR 3.5.30729)
```

Figure 4.5: Bro `software.log`, abbreviated for readability

Given that the user agent string is not required for the HTTP protocol, it is usually relatively easy for an end user to modify it (and it may, indeed, be necessary to access some services). Given that there is relatively little incentive for the average user to change the user agent string, and that it is even more unlikely in a corporate setting, the user agent can generally be trusted to contain accurate information. Since it is not essential to the standard, however, it is important to note that any user can change this string and not necessarily suffer service degradation.

Bro has built in support for recording the user agents of clients that it observes sending requests.

There is also an interesting special case for user agents over HTTP. Much modern software includes HTTP clients for doing external requests such as web API requests and update requests. In the example list above, the Valve Steam client implies that the Steam client from Valve software has been used by a client. This means that the client is likely being used

for playing video games. Such specialized software can, in other words, identify an asset's use cases instead of just identifying a web-browser being used for browsing.

4.2.3 Javascript libraries

The need for client side scripting in the html context has stimulated the use of JavaScript. While the server serves as the storage and distribution of JavaScript files, the scripts are executed in the client context.

There are a few usecases for knowing what JavaScript libraries are being served by your organization. From a maintenance perspective, it is better to keep the same libraries in use across products in the organization (see the Background section on Asset Consolidation). From a security perspective, knowing that the scripts you serve your clients are not tampered with is important in order to avoid malicious events happening due to files that you serve. In January 2014, Spiderlabs wrote about malicious file includes injected into JQuery library files on compromised servers. The files were then served to the visitors of webpages using these jquery scripts, which allowed for exploit kit installations on the visitors to affected webpages[11]. There is a tendency among javascript library developers to use and provide so-called minified javascript where content is obfuscated to save space. This technique, unfortunately, also allows an adversary to hide malicious code in a javascript file without arousing suspicion from casual inspection of the code.

Because the JavaScript libraries are served over the network, we can detect the transfers of these files, and identify them.

In Bro, it is possible to run md5 or SHA1 hashing over files transferred by loading `frameworks/files/hash-all-files` in the configuration. This allows us to have a previously generated master list of common JavaScript library files (or a custom one per organization generated using the asset detection system). By comparing the hashed sum of each file transferred, we can detect which known JavaScript files are being served.

4.2.4 Website whitelist

Most webpages are generally divided into different files, which are being served. Each file, such as an image, a document, JavaScript files or an executable can be detected during transfer.

We can use a modified version of the JavaScript solution above to detect malicious activity. Instead of having a known list of libraries and assigning asset information to each serving host, we can instead have a known list of hash sums for files being served, and trigger alerts for non-registered assets. Should a JavaScript file have been modified – say, to add a malicious include – it will trigger as a non-registered asset. Should an alarm notification be tied to this event, a compromise of the serving host can be detected. Essentially, we are leveraging asset information to detect non-standard behavior of our server, and defending based on that information.

Three things limit this technique:

- There is some computational overhead to hashing all files transferred from a larger website. To hash a file, all traffic related to the file needs to be read into memory and analyzed by the hashing mechanism.
- Building an authoritative list of hashes can be hard – it requires recording the states of each file at a known good state. This may require manual analysis.
- Keeping said list updated can be even harder, especially if the webpage is dynamic in nature – as many pages are these days.

One can limit the required computation by limiting the file types that have to be hashed and analyzed. For instance, it is possible to limit the hashing to only certain file types, at the cost of leaving some room for an attacker to transfer malicious files.

Building the list can be done either on the server, by hashing all served files manually or with scripts. Alternatively, one can use a Bro script to record URL / hash pairs, and simply browse the webpage. Care must be taken in ensuring that these activities are done while the files are in a known good state.

Regardless, such an effort can be well worth the effort required in order to avoid infecting visitors to webpages you are controlling.

4.2.5 Server header

The Server header is a field used by HTTP servers much in the same way as greeting banners are used by other protocols. The header is used for statistics and promotion of the web server being used.

To an asset detection system, knowing what server software is serving the current request is nice because many vulnerabilities will be specific to certain web servers.

A good source for examples of these headers is HackerTarget.com's data[1] from crawling the Alexa Top 500k pages. It is available as a zip file from their webpage. This data allows an analyst to take into account a wide variety of headers when making fingerprints.

```
1 Server: Google Frontend
2 Server: BigIP
3 Server: Microsoft-IIS/7.5
4 Server: Apache/2.4.6 (Unix)
5 Server: Apache/2.2.15 (Red Hat)
6 Server: Ning HTTP Server 2.0
7 Server: nginx/1.4.7
8 Server: Apache/2.2.20 (Unix) mod_ssl/2.2.20 OpenSSL/0.9.8e-fips
    -rhel5 mod_jk/1.2.28
```

Figure 4.6: Example server headers

The purpose of this header is to disclose the serving software. Some servers will also include server versions, or even the Operating System running the software. In figure 4.6 one server is also disclosing some of its modules and their versions.

A system administrator can usually turn off the header with access to the configuration files.

4.2.6 X-Powered-By header

Among many header fields used by HTTP servers is the non-standard field X-Powered-By, which is used by many servers to tell the client which technology is being used to serve the web application that is being requested.

This is a boon for the asset detection system. In essence, the serving framework for the web application is exposed to us. Given that many vulnerabilities exist in such frameworks, knowing what is used behind each webapp is very valuable to us.

```
1 X-Powered-By: UrlRewriter.NET 1.8.0
2 X-Powered-By-Plesk: PleskWin
3 X-Powered-By: PHP/5.3.27
4 X-Powered-By: Servlet 2.4; JBoss-4.2.2.GA (build: SVNTag=
    JBoss_4_2_2_GA date=200710221139)/Tomcat-5.5
5 X-Powered-By: ASP.NET
```

Figure 4.7: Example `X-Powered-By`

For the most part, this header will be set to a string pair of a framework name and a version number. As one can see in figure 4.7, some frameworks will not expose its version, while others will have a concatenated list of several frameworks.

This is a non-standard - but often used - header, and as such we cannot expect to see it in all HTTP responses. Since it is not known to serve any purpose except statistics and publicity for framework authors, system administrators may elect to modify and remove it at will. Modifying it to an alternate framework may be a simple attempt to create security through obscurity. Regardless, most instances of the headers are likely to be accurate.

4.2.7 Specific headers

Some frameworks also have their own specific headers. Much like the Server and X-Powered-By headers, there are some non-standard headers used by specific frameworks. A few can be seen in figure 4.8

We can make Bro scripts that pick up each of these fingerprints and assign the corresponding software to the originating IP.

Like all previous headers, these are not essential for the HTTP protocol, and thus carry the risk of them being modified or removed.

```
X-Powered-By-Plesk: PleskWin
x-aspnet-version: 4.0.30319
X-Mobilized-By: WordPress Mobile Pack 1.2.
X-Generator: Drupal 7 (http://drupal.org)
X-Drupal-Cache: HIT
X-Django-Cache: Yes
```

Figure 4.8: Example of software-specific headers

4.2.8 Filetypes

A final possibility for detecting which frameworks or execution environments a server uses is using the filenames for successfully served files. Many websites will have a file extension indicating what file type is being requested/served.

By using file extensions that are specific to certain execution environments (.php, .aspx, .pl), we can make a reasonable guess at which environment is being used by the server.

File extensions are relatively common in use, but in later years using htaccess and mod_rewrite rules (and other solutions depending on server software) to prettify URLs to not include file suffixes or change how URLs are parsed by the server. These techniques can also be used to modify the file-extensions to stymie anyone who wants to translate from file-extension to server environment.

In Bro, this can be detected using regular expressions over the URI of a given response.

4.2.9 Port Usage

Software that needs to be listening for incoming connections, usually servers, typically needs to be listening on specific well known ports. As an example, port `80/tcp` will usually be reserved for web servers as a class. Any connection by HTTP clients where only a hostname, and no port is not specified will use port 80 as an assumption. This is typical to most software where there is no dependency on a discovery or announcement protocol. These well known ports can be found organized by IANA , or on many unix systems in the file /etc/services. Ports are assigned in 3 categories:

- System ports (0 – 1023)
- User ports (1024 – 49151)
- Dynamic/Private ports (49152 – 65535)

Ports below 1024 will generally only be available for the root/administrator user on the machines running the software. These are often seen in connection to server software or services which run continually.

As is stated on IANA's webpages in a disclaimer: "THE FACT THAT NETWORK TRAFFIC IS FLOWING TO OR FROM A REGISTERED PORT DOES NOT MEAN THAT IT IS "GOOD" TRAFFIC, NOR THAT IT NECESSARILY CORRESPONDS TO THE ASSIGNED SERVICE". While ports will usually be a good indicator of what type of software is running on a server, it is often trivial to change what port a piece of software should be listening on. On the other hand, changes to ports will usually incur overhead of work for both the administrator of the server software as well as the client user, since reconfiguration must happen on both ends.

Two cases of Port usage detection have been implemented as examples for the proof of concept in the thesis.

The first is a general port registration script, which detects TCP streams which are established, and UDP streams that receive a response. These ports are considered open and listening. This detection gives a general indication of what software is in use on different IP addresses.

The second case is a script for registering software known to be operating on specific ports. A good example of this is the case of Dropbox, which is known to broadcasts on port `17500/udp`. Since this is a unique behavior, and the port is not known to be used by any other software, we can assign the software to the originating IP.

4.2.10 Registrations

Software that uses web-services as backends, such as Dropbox, will also have to register or check in with the servers to be able to use the services these servers provide.

Usually, this type of behavior will include logging in or providing some other means of authentication for services that requires this. Other services that are not personalized will instead see traffic requesting data from the servers, using methods such as web-apis or other ways of transferring the information required by the application. In the case of Dropbox, the registration is done using a web request to servers belonging to Dropbox.

```
1 #fields client   server  port    method  hostname        uri
2 10.0.0.16        108.160.163.100         80      GET     notify7
    .dropbox.com     /subscribe?host_int=##########&ns_map
    =########_##############,########_############,#######_
    ############,#######_###############,########_
    ##############,########_##############&user_id=#######&nid
    =######################&ts=##########
```

Figure 4.9: Dropbox registration request (numbers replaced with #)

It is important to be aware that this type of traffic also could be faked, and not only for malicious purposes. In Chat Wars[4], the author retells of his days developing the interoperability between MSN Messenger and AOL chat, and how they imitated the protocol used between AOL clients and AOL servers. A sufficiently good imitation would likely also fool an asset detection system.

4.3 Update

4.3.1 Update servers

With the advent of the internet, a lot of software went over to using the internet to update itself. Software will usually make a request to a server under the control of the developer of the software.

A solution to detecting software was presented in Mats Erik Klepsland's thesis "Passive Asset Detection using NetFlow"[10], which used the IPs that are used only for updates for specific software or operating systems to fingerprint the presence of that software.

By finding IP addresses or hostnames that are only used for specific update servers, we can separate out the clients connecting to them as having that software or operating system installed.

Unfortunately, there are several limitations to this. First and foremost, the approach is rendered moot the second a server is being used for software other than that which is expected by the script. In Klepsland's case, the operating system detection by update server had false positives, and he posits that it is likely to be because of hosts with Microsoft Office installed. In other words, since the update server was not dedicated to the single software we wish to detect, it is generating false positives. This is why this thesis does not recommend using this approach.

4.3.2 Update URLs

Given that we are using a NIDS system to capture and analyze network data, we also have access to packet payloads. This allows us to modify the update server solution above, and restrict the HTTP based update requests to specific software in order to avoid false positives. In most cases, updates will follow the same pattern as outlined in the package manager section under installation in this chapter. Some software update requests will however add more information in its requests.

Such requests will generally contain a software-specific request for a unified page where information about the newest software is available. Alternatively, it will inform the server of its version, and let the server provide information on the availability of updates.

```
1 #fields client  server  port    method  hostname        uri
          useragent
2 10.0.0.16       46.4.223.210    80      GET     sw-update.obdev
    .at     /update-feeds/littlesnitch3.plist       Little%20
    Snitch%20Software%20Update/4052 CFNetwork/673.0.3 Darwin
    /13.0.2 (x86_64) (MacBookPro11%2C3)
```

Figure 4.10: Little Snitch update check

In figure 4.10, the Little Snitch software requests a URL meant for all versions of the software. The webpage requested will serve an XML formatted file which the software will parse and then present the choice of updating to the user.

The request in figure 4.11, for the software Virtualbox, provides the server with a platform and version number. The page returns a link to

```
1 #fields client  server  port    method  hostname        uri
        useragent
2 10.0.0.254      137.254.60.34   80      GET     update.
    virtualbox.org   /query.php?platform=WINDOWS_64BITS_GENERIC
    &version=4.2.16_86992&count=4&branch=stable   VirtualBox
    4.2.16 <win.64 [Distribution: Windows 7 Service Pack 1 |
    Version: 6.1 | Build: 7601]>
```

Figure 4.11: VirtualBox update check

the newest version of the software, provided that it is not up to date.

Both cases can be used to derive which software is installed on the client. Additionally, the VirtualBox case provides information on the version of the software, as well the platform of the installed software.

Also to be noted in this case is that since the software is using a built in HTTP client to make these requests, they have also been helpful enough to provide us with a user agent, which in these cases sends the software version as well as the operating system. In the case of Little Snitch it also sends the hardware version of the physical machine it is running on.

When combining update URLs with user agents, it is possible to find software installations with a high degree of certainty. Like all other cases, it is possible that a client will modify traffic to inject false positives.

4.4 Hostname

A special case for asset detection is the detection of hostnames. While the hostnames themselves are uninteresting in that they do not prove the existence of any software or other asset information, they are useful in two ways.

First of all, the hostname is usually the more meaningful name for an asset behind an IP. It is more human friendly, and will usually follow a specific host, independently of new or reassigned IP addresses. In some organizations, the hostname will be the preferred method of identifying hosts, given that IPs will potentially change during a later DHCP renegotiation.

To a security analyst or a system administrator, the hostname can indicate the function of a host during manual analysis. While it is not evidence of a specific function, it will in many cases indicate the type of host or the function it serves.

4.4.1 HTTP Hostnames

When making HTTP requests a client will usually indicate a specific hostname where the resource is located. Usually, this hostname is found in the Host header.

According to the RFC for the HTTP 1.1 standard if the URI is not an absolute address, the host designated by the Host field must be a valid host on the server. If the host is not valid, the server should return a 400 (Bad Request) error.

This allows us to gather hostnames from all successful HTTP requests.

There are some caveats to this, though. The HTTP hostname does not necessarily correspond to a network hostname. When using a reverse proxy in front of several HTTP servers, the proxy will respond to requests for all hosts it proxies. To the asset detection system, it would seem that the proxy should be assigned the hostnames, while the proxied servers are actually the ones the hostname should correspond to in the system. In other words, the hostname will often correspond to the canonical server hostname, but this will not always be the case.

Another limitation is that several common servers are not totally compliant to the RFC as above. Some HTTP servers will redirect the request to a default server if the request hostname does not exist on said server, and thus provide us with a success code instead.

In the example implementation made for this Master's project, a script has been set up to record Host and IP pairs on successful requests to a server over HTTP.

4.4.2 DHCP

DHCP is the protocol for dynamically distributing several key configurations upon connecting to a network. One of the more important functions is requesting/receiving an IP address when connecting. The protocol runs on UDP and utilizes broadcasts of packets.

DHCP is interesting because it allows the client to specify its hostname, and inform the DHCP server (and other listening network nodes) of it. Such information is optionally included in a DHCP option field.

This information is useful because it tells the network (and the asset detection system) the hostname by which the client identifies itself.

The limitation to this is that the client can choose its own hostname. Unlike a DNS entry, it is not (or should not be) canonical to the rest of the network. Additionally, as the DHCP is generally a local function, it will not propagate outside its own network. This means that in certain network topologies, such as the topologies often seen by ISPs and MSSPs, scripts utilizing this information will be useless.

We can detect these settings in Bro by receiving certain events produced by the DHCP analyzer module of Bro.

4.4.3 DNS sniffing

A final, and more canonical look upon hostnames in the context of an organization is the responses to DNS queries. The Domain Name System allows for a translation from hostnames to IP addresses.

In our case, these hostnames are interesting whenever those using the asset detection system do not have access to the DNS being used in an organization. This is, for example, often the case for managed service providers. By passively detecting DNS records and their assignments, we can build a list of domain names used in an organization.

A limitation to this approach is that a malicious client could use a malicious or misconfigured DNS server to fill asset records with bogus domain names. Additionally, since most DNS traffic uses UDP, traffic can be

spoofed to appear to come from an authentic DNS server.

Bro contains a DNS parser, which can be used to parse DNS responses.

4.5 Proxy/NAT detection

Another special case for asset detection is detecting IP addresses that are either proxy or NAT addresses.

A proxy server is a server that acts as an intermediary between clients and servers. A client will generally connect to a proxy, which will then relay the request to a server. The response is then carried back to the proxy, which relays the information to the actual client. Most proxies seen today are web proxies, which relay HTTP requests. In a corporate environment, web proxies are usually used to speed up requests by caching files, filter data depending on the organizations policies on web surfing, and to allow for security functions.

The concept of NAT addresses are most often used today for the purpose of IP masquerading, that is, to hide a network of IPs behind a single IP. Most home routers used today are used as NATing devices, hiding a private network behind a single public IP.

Both of these concepts are important to an asset detection system because they allow a single IP to "hide" traffic coming from multiple distinct networked hosts. A single host behind one of these systems may have updated versions of software, while other hosts in the network are using vulnerable versions. In cases where a malicious attack targeting these vulnerabilities are detected, it is important for the analysts to know that several hosts reside behind that single IP, and that they have vulnerable versions of the software – as opposed to thinking that the different records of software versions are a product of – for instance – a host being updated from the vulnerable version of said software to a newer version.

4.5.1 X-Forwarded-For

X-Forwarded-For is an HTTP request header inserted by various proxy solutions. The header is not a standard header (as denoted by the X prefix),

but is implemented in solutions such as Squid, Bluecoat and others.

Since the header is typically only present in communication going through a proxy, this information allows us to identify an IP addresses where it is a proxy, or one of the clients NATed behind it is a proxy.

On the originating host it is used to identify clients behind a proxy by the end server – negating the anonymization implications of using a proxy, while still allowing the use of proxies for other reasons.

For the asset detection system this header is useful because it allows for identification of proxies, and because it can give indications of the network addresses used behind the proxy.

Like several other techniques presented previously in this chapter, scripts to detect this behavior will only have to consider HTTP headers. As such, it is relatively trivial to implement. The presence of X-Forwarded-For indicates a proxy host, while the IP(s) set by the header can optionally be used to map out the network behind the proxy IP.

4.5.2 Proxy specific headers

Much like the behavior of some software and frameworks described previously in this thesis, some proxies have their own specific headers. These are usually appended to the list of headers to support specific functions in the software.

In the case of the proxy vendor Bluecoat, there is use of a specific header: X-Bluecoat-Via. This header is used to detect loops in the network, potentially stopping a circle of proxies forwarding to each other .

The presence of this header can be used by our asset detection system to indicate the presence of a Bluecoat proxy behind an IP.

4.5.3 Mutually exclusive OS/Software

A final effort to detect proxy or NAT hosts is to leverage previously gathered asset information rather than just the traffic passing through cur-

rently. By finding the use of mutually exclusive asset information, we can infer when different systems are behind an IP.

One method of detecting reverse proxies behind IP addresses is finding different server software using the same ports. Port `80/tcp` (HTTP) simultaneously serving traffic from both an apache and an IIS server is a sure indication that there is some form of reverse proxy being used in front of multiple servers.

Another method, this one for detecting NATed or proxied hosts is by detecting the presence of several mutually exclusive Operating Systems. Both linux and windows user agents are a good indication that either the host is used as a NATing host, proxy, or that a client is spoofing traffic.

Because this detection method is not done over network data, but rather over gathered asset data, detection like this should not be done in Bro, as that would cause too big of a burden memory-wise. In a system like the example implementation done in this thesis, the process inserting new asset data into the database backend would have done this, as much of the information will be looked up at insert time anyways. Alternatively, this effort could be done asynchronously by a separate process dedicated to iterating through the database and detecting mutually exclusive OS or software.

Chapter 5

Evaluation

5.1 Method of evaluation

The goal of the evaluation was to test if the proof of concept based on the techniques described in this thesis could effectively detect assets. This chapter is dedicated to investigating the research question regarding how gathering asset data works in practice. To this end, I have chosen to evaluate the system in a network that is largely unknown to me.

5.1.1 Evaluation environment and execution

The data used to evaluate the efficiency of the system created during this project was taken from an office network. The traffic was captured over 72 hours, from noon on a Wednesday to noon the following Saturday.

The dataset is approximately 280GB of dumped traffic, divided into 2GB files. Each of these files correspond to a data point on the graphs following in this chapter. The difference in rate of data transfer during different times of day is why there is variable spacing between points on the graphs.

All traffic was captured using the program `dumpcap`, running on a computer with traffic spanned from a switch in the evaluation environment. The environment is known to be mixed OS-wise, containing hosts using both Linux and Windows.

The asset collection sensor was configured to only detect assets belonging to the client IP subnet. This way, asset information for external hosts

is not generated, and the task of verifying the asset information is made simpler. It is also important to note that there is a varying amount of detail in the fingerprints, and that therefore new entries in the database are generated when a signature detects differing levels of detail. This is especially easy to identify in the case of OS detection, where many of the signatures are limited to only identifying the OS family (Windows/Linux/OS X), while other fingerprints identify version as well. The signatures used in the evaluation can be seen in table 5.1.

The traffic was replayed to the sensor over an interface to get a realistic look at the performance of the rules implemented. The traffic was replayed using the tool `tcpreplay`, which allows replay of packet dumps over a network interface at specified speeds. Using a set speed of 150Mbps, the sensor had minimal packet loss (less than 0.3% at its highest).

Fingerprint name	Capability
assets_apt-get.bro	Detects ubuntu/debian (and some other) versions from update urls using distro codenames.
assets_apt-get_update.bro	Detects Debian derived linux package installations using apt-get.
assets_fedora.bro	Detects fedora version from update checks.
assets_fedora_update.bro	Detects fedora package installations using yum.
bro_softwaredetect.bro	Bridges with Bro's built in user agent and server banner facilities.
dns_hostnames.bro	Detects hostname assignment from DNS responses.
dropbox_broadcast.bro	Detects dropbox LAN sync communication. Not useful in evaluation due to dropbox not being installed in environment.
dropbox_register.bro	Detects dropbox registering to web-server.
http_banner.bro	Saves Server: headers from HTTP responses
http_hostname.bro	Registers hostname from successful HTTP requests.
js_hashcheck.bro	Detects versions of jquery being transferred using hash. Not useful in this test, as all javascript files were cached (returned 304 Not Modified).
ports.bro	Detects port usage under port 1024.
windows_onlinecheck.bro	Detects windows installations from onlinecheck.

Table 5.1: Fingerprints used during evaluation of the proof of concept

5.2 Results

In all the following graphs, we see the initial learning phase of the system as it comes online. The reason for this is the same as was seen in Klepsland's thesis[10]: that the system is initialized and immediately detects many of the hosts which continuously communicate.

While the same hosts may continue to be detected by the system over the course of the sensor's running time, they will not be registered as new asset. As assets are found over time, the rate of new entries will decline until there are changes in the environment, such as new hosts, services or updates.

5.2.1 Host detection

The host detection rate is based on the other types of asset information. A host is not registered in the system until there is tangible information about it. The times at which new hosts were detected by the system can be seen in figure 5.1.

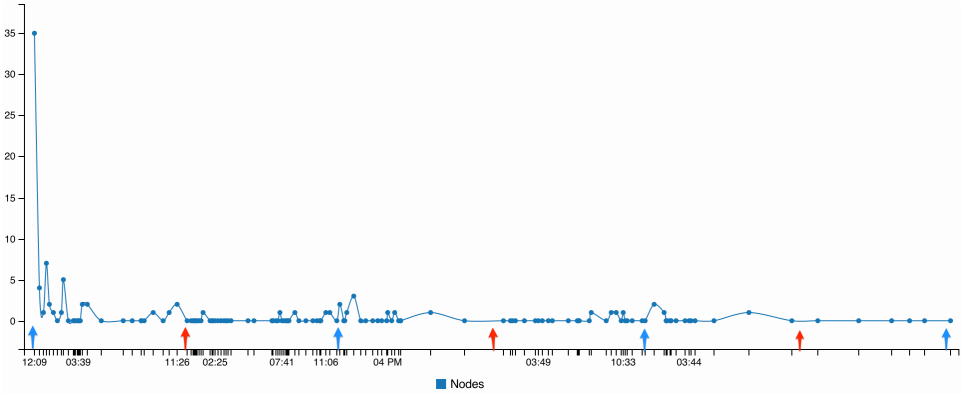


Figure 5.1: New hosts detected over 3 days, red arrows indicate midnight, blue arrows indicate mid-day

The new entries for hosts are clustered around the working hours on the working days, which is reasonable given that this is the time when non-automated software is communicating over the network.

In total, 84 hosts were detected during the evaluation period of the tool. This includes some servers in the IP range as well as clients.

5.2.2 OS detection

The OS detection is based on 3 different techniques spread over 5 fingerprint rules. A total of 43 entries spread over 16 hosts were found.

28 entries were based on the `softwaredetect.bro` rule, based on finding indicators of client OS using strings found in user agents reported by Bro.

10 entries were based on software updates for Ubuntu and Fedora, detected by `assets_apt-get.bro`, `assets_apt-get_update.bro` and `assets_fedora_update.bro`

A final set of 5 entries were based on `windows_onlinecheck.bro`, which detects Windows hosts connecting to the domain Windows hosts use to detect online connectivity.

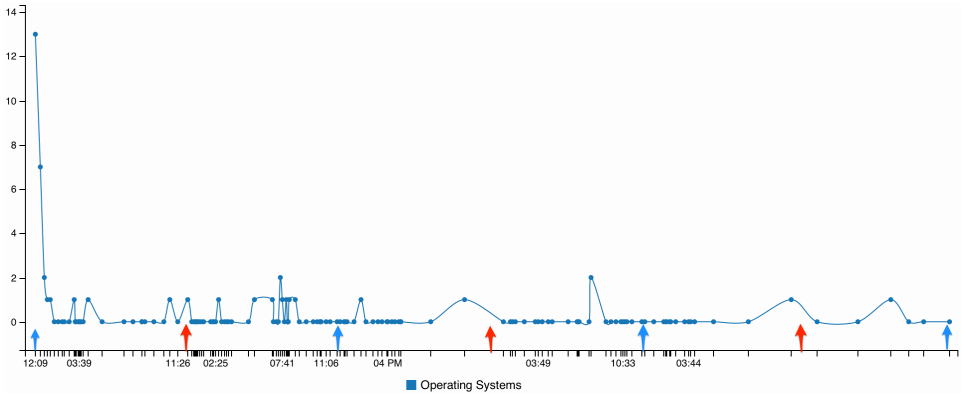


Figure 5.2: New Operating systems detected over 3 days, red arrow indicate midnight, blue arrows indicate mid-day

Beyond the initial spike in the learning period, the statistics show concentrated instances of new operating systems being seen in the morning hours on the second day, as well as a small spike on the third day. The final two detection instances are around midnight and 8 in the morning, and coincide with shift changes, where new software with better indications of OS versions were detected.

5.2.3 Software detection

Software detection is based on 4 fingerprint rules. A total of 317 entries spread over 50 hosts were found.

20 entries were based on `http_banner.bro`, which operates using the `Server:` header field in HTTP communication. This script overlaps with the detection capabilities provided by the Bro software framework.

193 entries over 48 hosts were found based on `bro_softwaredetect.bro`, which detects user agent strings as well as server banners.

26 entries were found based on `assets_apt-get.bro`. These were for a single update and correspond to the spike seen on the third day.

78 entries were found based on `assets_fedora.bro`. These were for a two machines doing updates/installations and the larger of these installation sessions correspond to the spike seen on the final day.

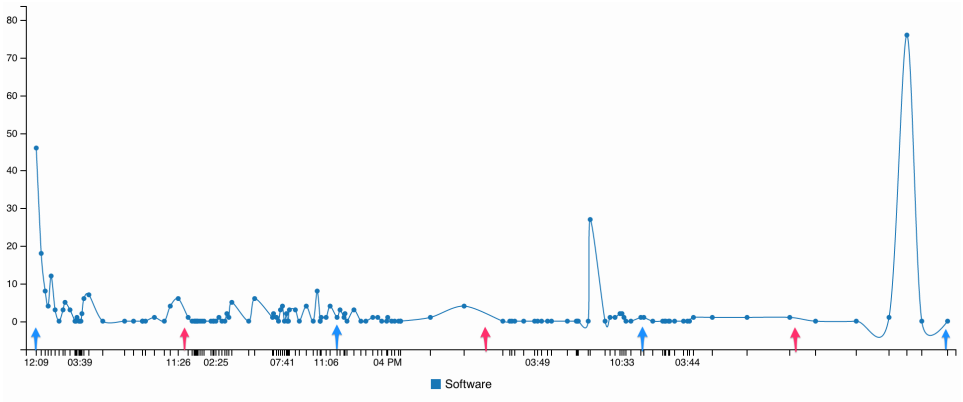


Figure 5.3: New software detected over 3 days, red arrow indicate midnight, blue arrows indicate mid-day

As with the OS statistics, we see clusters starting around 8 o'clock in the morning, continuing out the working day.

Additionally, as mentioned above, there are two big spikes in detected software in the morning on friday and saturday. These are software updates, and operating systems such as Windows and some linux distributions are known to do automatic updates at night by default if enabled. This lets the machines avoid using time and network bandwidth during

operating hours. Two spikes during the first night night are also attributable to updates, there due to user agents related to Windows updates.

One lesson which can be drawn from this is that the some of the most informative asset information is only available sporadically. Should the timespan of surveillance be expanded, we can expect more software updates to improve our asset information.

5.2.4 Hostname detection

Hostname detection is based on 2 different techniques, DNS and HTTP host header. A total of 79 entries spread over 57 hosts were found.

58 entries were based on `dns_hostnames.bro`, over 51 hosts. As the name implies, this script detects DNS responses and assigns hostnames accordingly.

21 entries based on `http_hostname.bro`, over 15 hosts. This signature detects successful HTTP requests, and records the `Host:` header used by the requesting client.

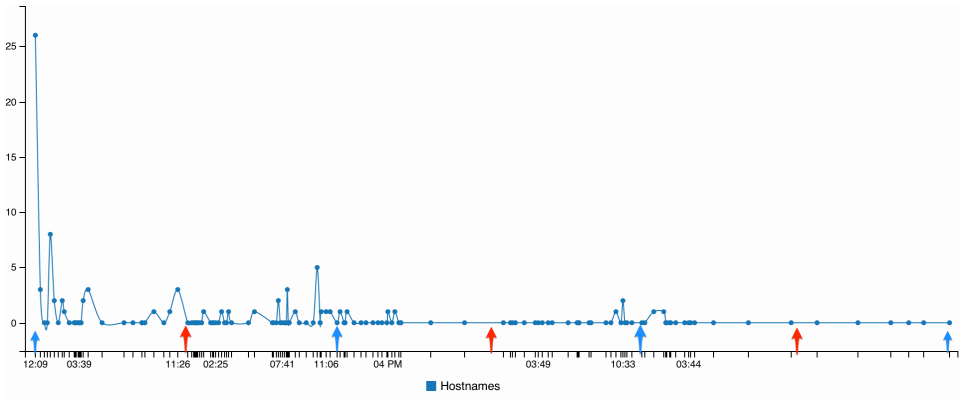


Figure 5.4: New hostnames detected over 3 days, red arrows indicate midnight, blue arrows indicate mid-day

The detection of hostnames were, as with the other classes of detected assets, mostly driven by traffic generated during working hours in the environment.

There is some traffic related to automated traffic during the night hours

the first night, but no more is detected in later nights due to the hostnames already being discovered by then.

5.2.5 Port detection

Portname detection is based on `ports.bro`, which detects tcp and udp ports under `1025` responding to connections. A total of 88 entries spread over 50 hosts were found.

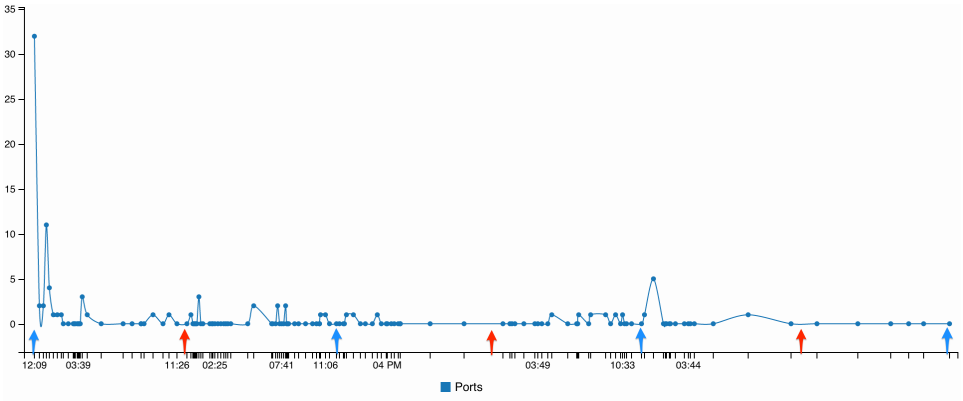


Figure 5.5: New ports detected over 3 days, red arrows indicate midnight, blue arrows indicate mid-day

Like all the other categories, new port detections also generally only occur during working hours.

5.3 Data presentation

As part of the proof of concept program made for this project, I made a web interface to look up asset data gathered based on individual hosts. The web interface consists of a simple lookup form where one can enter the unique id for the asset being we wish for asset information about, and a display where asset information is presented for the requested hosts upon submitting the form.

The information is presented in a table view, with most of the fields from the asset database exposed for each entry. The only field which was not exposed here (beyond entry id) was the raw data on which the initial entry was based. It is likely desirable that this information is available to

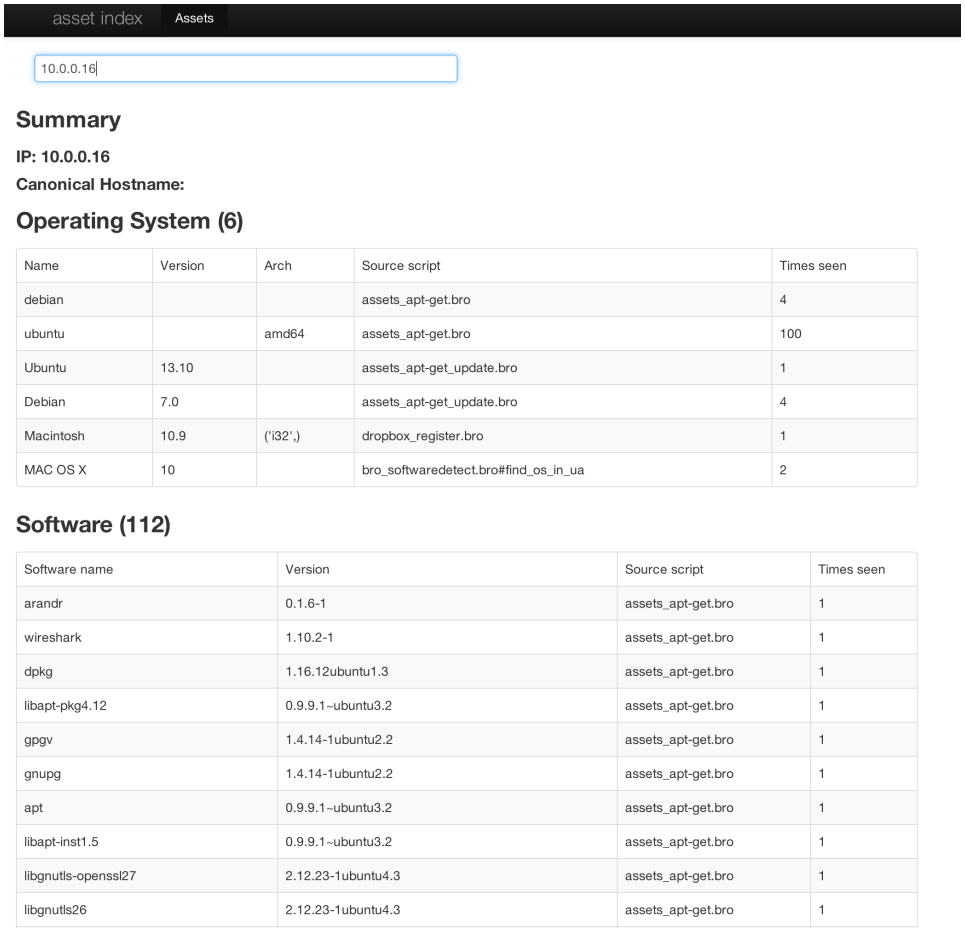


Figure 5.6: Asset detection results presented in a web view

analysts in a more developed system.

An example lookup in this can be seen in figure 5.6. The figure shows a host running 2 virtual machines. The Ubuntu virtual machine has in this case also downloaded updates from the internet, and these have been registered to this asset as installed.

The asset viewer is part of a multi-mode script developed for the proof of concept. The script, `assetweb.py`, has three modes.

The first mode, invoked with `python ./assetweb.py -mode makedb`, initializes a sqlite database based on the schema described in the script. This database is stored as `assets.db` in the current working directory, and is where the asset data is stored.

The second mode, invoked with `python ./assetweb.py -mode bro`, runs the script in a mode which connects to a local Bro instance using the library `broccoli`, which allows for sending and receiving events from Bro. This is the mode in which asset data is transferred from events in the Bro engine to the database.

The third mode, invoked with `python ./assetweb.py -mode web`, results in a web server being run on port `5000/tcp` on the local host. This serves the web interface seen in figure 5.6, as well as an API serving json formatted asset data.

Chapter 6

Discussion

6.1 Quick PRADS comparison

PRADS is a passive asset detection system, designed along the same lines as p0f, which was mentioned in the Background chapter. PRADS has support for OS detection through TCP/IP stack fingerprinting, as well as support for OS version detection through `Server` strings.

When compared to the system shown in the evaluation chapter, PRADS has a marked advantage in the cases of detecting OS versions installed on servers. The server OS identification through server banner was not implemented in the proof of concept ruleset, which led to no OS being identified for any of the dedicated servers encountered. The information was stored in the host banner, and in cases where the OS or distribution was leaked in this variable it was still available to an analyst in the asset system, but the machine was not marked as an host using a specified operating system.

The disadvantage for PRADS lies in some inaccuracies in OS, such as a Windows 7 host which was identified as a Windows XP/2000 and an OS X machine being identified as an iPhone. There were also instances where PRADS ended up with unknown OS fingerprints, but where the evaluated system detected OS version from the user agent strings and from Windows checking for the Microsoft online connectivity check server. Since PRADS uses packet fingerprinting, the linux hosts were only identified by a kernel version - and in the case of this dataset only by "Linux 2.6". On all hosts identified to a specific distribution release - where kernel version is available - the machines were running a later kernel than Linux 3.0.

One definitive advantage to PRADS is that it tags what ports a machine is a client for, which is not done by the prototype system. This type of information could be useful to profile the host type.

6.2 Comparison to proxy/HTTP logs

Of the fingerprints detected by the asset detection system, many could potentially have been executed with log analysis of HTTP related logs instead of IDS analysis. Log sources that could provide this type of information are not unusual, and can be provided by solutions such as most proxies, Bro IDS or Suricata.

These logs usually include both user agents for the clients, as well as URLs requested. Due to the vast amount of data that is transferred over HTTP these days, analysis of these logs can provide us with much of the same asset information as that seen implemented in the Implementation chapter of this thesis.

A major advantage of using these types of logs is that we have the possibility of analyzing data from many alternate sensor types. Many organizations do not necessarily have the wish or the resources to deploy an IDS for asset detection purposes, but the use of proxies is widespread. By using the proxy-related logs organizations can get asset detection technology at a lower cost than what may initially be assumed.

Another advantage of using this type of analysis is that it can be done offline, and can easily be parallelized for performance. Since this analysis is done across logs, historical data can be gathered and analyzed at later dates. This includes using new signatures on older data to gather asset data from times when the rules were not created.

By moving to only analyzing data from these types of logs we also avoid some of the privacy implications of analyzing more data than necessary in the payload of packets. While plain URLs do have privacy implications too, much of what most people consider to be sensitive information is usually the data which is transferred in the body of messages. Examples of information that would not be available to the log-based utilities mentioned above, but available to the payload-scrutinizing systems, is the content of

HTTP POST messages and unencrypted web pages.

By moving the analysis to these types of logs, we do however lose some of the possibilities of gathering data from the content in the payload of the packets sent, but we still have the possibility of employing many of the same asset detection rules presented in this thesis in other environments as well.

6.3 Learning another language to make use of the tool

An issue with using Bro for asset detection is that an analyst wishing to make signatures will need knowledge of the domain-specific language used in Bro scripts. While the language is not very different from other programming languages, the use of a domain-specific language can be an additional hurdle to overcome to someone interested in this type of asset detection system.

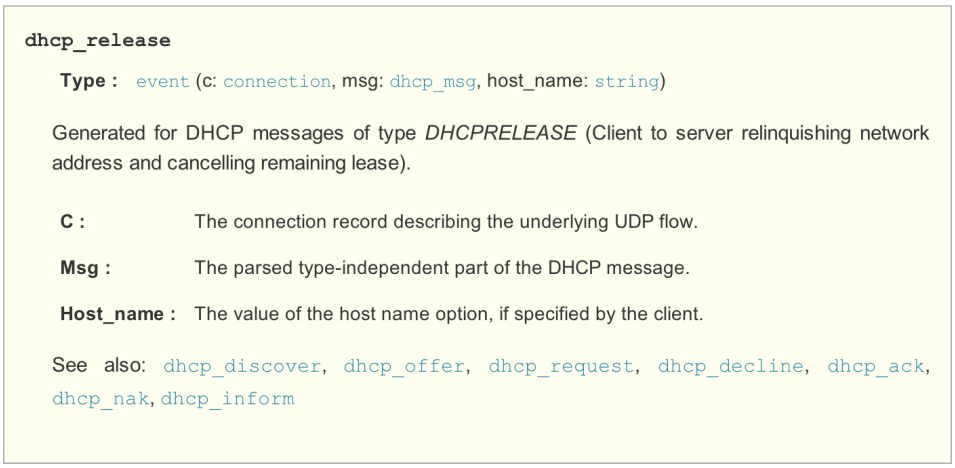
This problem is offset by good documentation of the use and development of Bro scripts found in the Documentation section on the Bro webpage. There is a basic introduction to the features of the programming language for Bro, accompanied by several examples of bro scripts.

There is also a section listing the events that are generated by the different analyzers present Bro, along with documentation on the data types and structures used by them. An example of this documentation can be seen in figure 6.1. The availability of this type of information, along with easy lookups of data structures - such as the `connection` structure seen in the example - by means of hypertext links, the process of writing new rules is greatly simplified.

6.4 Decaying Asset information

Another concept which has not been taken into account so far in the thesis, but which should be considered when implementing a passive asset detection system is the decay of asset information.

Because the network and the assets running on it is dynamic, we must



dhcp_release

Type : event (c: connection, msg: dhcp_msg, host_name: string)

Generated for DHCP messages of type *DHCPRELEASE* (Client to server relinquishing network address and cancelling remaining lease).

C : The connection record describing the underlying UDP flow.

Msg : The parsed type-independent part of the DHCP message.

Host_name : The value of the host name option, if specified by the client.

See also: dhcp_discover, dhcp_offer, dhcp_request, dhcp_decline, dhcp_ack, dhcp_nak, dhcp_inform

Figure 6.1: Example of bro event documentation for dhcp release event

take into account the fact that data about our assets can and will become outdated.

The dynamic nature of computer networks and assets on them manifests itself in several ways, which must be taken into account by a user of asset detection systems.

6.4.1 Changes in assets

The primary way the dynamic nature of computer networks and the assets on them has to be taken into account is the possibility of changes in configuration, software or even hardware.

Software updates, as an example, are usually encouraged or even mandatory to users of workstations in places that have a security policy.

Depending on the software used by the organization, and the update schedule for said software, a limited time can pass before an asset database must be considered to be out of date.

6.4.2 Changing IP address

Dynamic DHCP also affects the asset inventory list. In most networks the use of an IP address as a primary key in the asset database would be the natural choice, but should IP addresses be handed out randomly for assets

we wish to track, then it might be necessary to use alternative unique identifiers to tie together information. What such IDs should be is dependent on the organization, sensor placement and other factors, but some potential alternatives that may fulfill the requirements are MAC addresses and hostnames.

6.4.3 Timestamps, versions and decay

One potential solution to defeat normal changes in the network is to record timestamps for each registered event containing asset information. By having a record of when an asset was last seen, the information can be allowed to decay, and automatically prune such information when it is thought to be outdated.

The policy for such decay will vary from organization to organization, depending on how quickly asset information is thought to change under normal circumstances. As an example of this, update strategies for Operating Systems and software will vary from organization to organization, depending on needs. While some organizations run updates as they become available, other organizations may employ strategies where they test updates for a limited amount of machines before rolling them out organization wide. Another alternative is to stagger the updates between fallback environments, ensuring that if there is a critical bug or change, both primary and secondary environments are not disabled.

Another possibility is to use the upgraded software version numbers being registered to the same assets as an update of the software, and remove or update the previous records. When IP addresses are used as the unique identifier in an asset detection database, it is important that such deletions are not done to assets where there is a potential for the IP acting as a proxy or NAT address.

6.5 Opportunistic Security and general encryption

An issue that is rapidly gaining relevance with the increased demands of security for the Internet is the use of HTTPS and other technologies to provide end-to-end encryption for client/server connections.

Efforts, such as the HTTPS everywhere plugin[7], encourage users to use HTTPS for most connections to ensure that encryption (and other HTTPS provided security services) is provided immediately upon connecting to servers.

Other efforts, such as the standardization process for HTTP/2[6], have considered adding mandatory encryption to all implementations using the standard. In a response to concerns of pervasive monitoring of internet traffic, there have been published drafts on the concept of opportunistic security in the forum of IETF [8]. These drafts specifically mention opportunistically encrypting connections when both endpoints allow for it.

While all of these efforts are likely to be good for the security of most of the Internet, it also presents a problem to implementations of network traffic sniffing for benign purposes, like network intrusion detection for previously unencrypted traffic – and of course for systems like that which is described in this thesis.

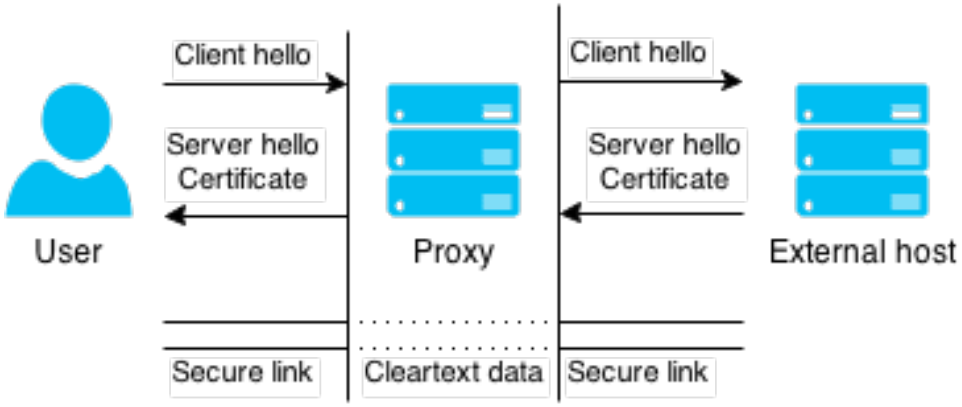


Figure 6.2: Simplified SSL proxy communication diagram

As can be expected, possibilities for removing encryption from communication streams are limited. In larger, managed environments it is possible to make use of trusted certificates and proxies like those delivered by BlueCoat[5] to intercept encrypted communication. By intercepting HTTPS requests, the proxy can take over as the man-in-the-middle (see figure 6.2), negotiating the traffic between client and external server. The proxy negotiates the HTTPS encryption to the external server, but provides its own certificate to the internal client. The proxy can then decrypt the data received from the server,provide security sevices on the content of the data

stream, before re-encrypting the data stream for delivery to the client.

6.6 Privacy - in memory analysis vs. logs

As mentioned previously, it is possible to do much of the same analysis described throughout this thesis by analyzing common log entries made by proxies or IDS solutions like Bro and Suricata.

This does, however, have some privacy implications. Using custom Bro scripts to do analysis on the traffic directly allows us to avoid storing these logs over longer time periods – or storing them at all. All sensitive data is only present in the memory of the sensor for a limited time until the data of the transfer is purged. HTTP logs are especially vulnerable to this, as whole web surfing sessions are stored and searchable. Such metadata allows for profiling of individual web surfers, and will by many be considered to be highly intrusive.

Instead only data that is relevant to the asset detection database should be kept over longer periods.

Chapter 7

Conclusion and future work

7.1 Goal fullfillment

Several research questions were presented in the introduction of this thesis. While I am hopeful that the information in the chapters following the introduction, I will also summarize my findings in this section.

- Q1) How can we design an asset detection system based on an IDS, such as Bro?
- A1) This thesis has explored and presented several methods of gathering fingerprints. In chapter 3, several methods of gathering network-based fingerprints for assets was presented, demonstrating some of the potential uses of information available to an IDS-based asset detection system.
- Q2) How practical is the process of collecting and using fingerprints in an IDS context?
- A2) In the Implementation chapter of the thesis, using the software life cycle, we demonstrated a range of categories where fingerprints could be found. The implementation presented in chapter 5, based on a selection of these fingerprints, shows that it is possible to design a usable system for asset detection and management of this type of information. We have shown that we have access to a wider variety of information than most other passive asset detection systems, by utilizing the additional information available to IDS systems. How labor intensive the research and development of signatures are will vary, from very specific signatures only able to cover a single type of software to signatures able to detect all software installed on a host

like the `assets_fedora.bro` script included in the appendix. The return of investment will of usually be lower on the former type of signature.

- Q3) What is the practical possible coverage of this type of system?
- A3) While the proof of concept system was limited in its coverage of assets, it demonstrated the potential for increased coverage when compared to classical passive asset detection. The possibly best illustration of the potential coverage which can be attained using IDS-based asset detection of software was shown in the rules for parsing `apt-get` and `yum` installations, which can potentially cover up to 100% of software installations using these systems. In most cases, the only likely omissions will be installations using offline media, compiling from source code or software installed prior to the asset detection was started.

7.2 Future work

As will often be the case when finishing a Master's project, I have discovered several ways in which the studied concept could be expanded and improved. This section lists some of the ways the use of an asset detection system based on payload data can be expanded and improved.

7.2.1 Expanded ruleset

Due to the time limitations for my Master's project, a full set of signatures for all software could not be defined. It is in the nature of a dynamic software ecosystem like that we have today that signatures will either be outdated or cease to exist in the first place, as new software versions and entirely new software is released.

Any asset detection system will likely need continuous development as new software is made available to the organization. Much like with Network IDS signatures, the ruleset will need expansion as new assets are required to be detected.

In the case of this project, there was only time to make a subset of rules as examples of what can be detected. Beyond the general lack of coverage for all software, the developed ruleset is lacking the presence of signatures

for specialty hardware like printers, network attached cameras and specialty software, which was not available to me.

7.2.2 Correlation rules

One type of rule that only got limited coverage in this thesis was the concept of correlation rules. By this I mean rules that use data points gathered over time to make assumptions about assets. In the Implementation chapter the concept was suggested for detecting NATed networks through mutually exclusive assets such as Operating Systems.

Other rules of this type can be very interesting, and should be subject to deeper investigation. One concept that could prove useful is making profiles of ports that are open by default in different operating systems. This is mostly specific to windows, where several services are on by default, but could also prove interesting to detect specific hardware that may have sets of odd default ports too.

7.2.3 Tools for rule creation

Another sorely lacking area is tools to analyze network traffic and to make signatures from them. While a few possible tools were presented in the design chapter of the thesis, I could not find a tool dedicated specifically to making the kind of signatures necessary for asset detection.

A tool which could compare and contrast network traffic between sessions, such as URLs requested or ports and hosts communicated with could be invaluable, not only to asset detection systems but also other purposes like gathering malware indicators and other indicators of compromise.

Another tool that could be helpful for projects like that presented in this thesis is the development of a unified asset detection rule language. This would allow a shared ruleset, even with different engines to detect the data. As mentioned in the discussion chapter, it is possible to do many of the things done in this thesis over logging sources instead, and a unified ruleset that could be used in both types of tools would ensure that organizations with different toolsets and data sources could still share the effort required to make an adequate sets of rules.

7.2.4 Ruleset validation

There is also a lack of tools and data sets to develop and validate rules. Many of the publically available datasets are scrubbed of payload data and anonymized before release to the public.

Public pcap captures of real or simulated network along with an accompanying list of hardware, software and other asset information could be very useful for following research and development. However, one must be aware that such captures will only remain relevant for a limited time before updates to software and hardware will render much of the data about components in the captured network as no longer representative of real networks.

Preconfigured sets of virtual machines covering different OS installations can also be very useful for the same purpose, giving an interactive and modifiable network of hosts for a researcher to investigate. The practicality of shared sets of virtual machines like this may be limited, given licensing requirements for commercial OSes like Windows and OS X.

7.2.5 IDS correlation

As has been suggested in the Background chapter, it is possible to use the data gathered as a part of an asset detection system to configure or optimize the ruleset used in an IDS solution, or alternatively to escalate IDS events when the combination of rule and asset information makes it more likely to be a possible event. Given how many false positives IDS systems may have, it is important to reduce the amount of events that must be analyzed by the limited amount of analysts available.

Bibliography

[1] *500K HTTP Headers | HackerTarget.com*. URL: https://hackertarget.com/500k-http-headers/ (visited on 31/07/2014).

[2] Alin Andrei. *Pipelight: Use Silverlight In Your Linux Browser To Watch Netflix, Maxdome Videos And More Web Upd8: Ubuntu / Linux blog*. URL: http://www.webupd8.org/2013/08/pipelight-use-silverlight-in-your-linux.html (visited on 01/08/2014).

[3] *Attack Surface Analysis Cheat Sheet*. URL: https://www.owasp.org/index.php/Attack_Surface_Analysis_Cheat_Sheet (visited on 21/07/2014).

[4] David Auerbach. *Chat Wars*. URL: https://nplusonemag.com/issue-19/essays/chat-wars/ (visited on 31/07/2014).

[5] *Bluecoat Knowledge Base - How to configure SSL proxy to intercept HTTPS traffic for an explicit deployment using a self-signed certificate?* URL: https://kb.bluecoat.com/index?page=content&id=KB5500 (visited on 31/07/2014).

[6] HTTPbis Working Group. *HTTP/2 Frequently Asked Questions*. URL: http://http2.github.io/faq/#does-http2-require-encryption (visited on 21/07/2014).

[7] *HTTPS Everywhere | Electronic Frontier Foundation*. URL: https://www.eff.org/https-everywhere (visited on 31/07/2014).

[8] IETF. *Opportunistic Security as a Countermeasure to Pervasive Monitoring*. URL: http://datatracker.ietf.org/doc/draft-kent-opportunistic-security/ (visited on 21/07/2014).

[9] CVE Initiative. *CVE - Frequently Asked Questions*. URL: https://cve.mitre.org/about/faqs.html (visited on 11/08/2014).

[10] Mats Erik Klepsland. *Passive Asset Detection using NetFlow*. URL: https://www.duo.uio.no/handle/10852/9044 (visited on 21/07/2014).

[11] Rami Kogan and Ben Hayak. *Beware! Bats hide in your jQuery!* URL: http://blog.spiderlabs.com/2014/01/beware-bats-hide-in-your-jquery-.html (visited on 21/07/2014).

[12] *Nmap - Free Security Scanner For Network Exploration & Security Audits.* URL: http://nmap.org/ (visited on 21/07/2014).

[13] Robin Sommer. 'Bro: An Open Source Network Intrusion Detection System.' In: *DFN-Arbeitstagung über Kommunikationsnetze*. 2003, pp. 273–288.

[14] *The Official Nmap Project Guide to Network Discovery and Security Scanning*. URL: http://nmap.org/book/man-performance.html (visited on 21/07/2014).

[15] Ryan Trost. *Practical Intrusion Analysis: Prevention and Detection for the Twenty-First Century*. Addison-Wesley Professional, 2009. Chap. Intrusion Detection Systems. URL: http://my.safaribooksonline.com/book/networking/intrusion-detection/9780321591890/two-signature-writing-techniques/ch03lev3sec1.

[16] Vijay Vaishnavi and Bill Kuechler. 'Design Science Research in Information Systems'. In: 2004. URL: http://desrist.org/desrist/content/design-science-research-in-information-systems.pdf.

Appendix A

Explanation of acronyms and expressions

Word	Definition
Asset	Something of value to an organization. Can be computer hardware, communications facilities, buildings, information / data or software.
Bro	An open source IDS system.
DHCP	Dynamic Host Configuration Protocol.
Fingerprint	An indicator of an asset's identity.
HKEYS	Entries in Microsoft Widows' registry
IDS and IPS	Intrusion Detection System and Intrusion Prevention System.
IOC	Indicator of Compromise.
ISP	Internet Service Provider.
ISS	Internet Security Systems. Produces several security products, including IPS products.
MAC Address	Media Access Control address.
MSSP	Managed Security Service Provider.
NAT	Network Address Translation.
Network node	A specific device on a network.
NMAP	Network Mapper, an active network mapping tool.
OSI model	Open Systems Interconnection model, a model describing network layers.
OWASP	Open Web Application Security Project.
p0f	Passive traffic fingerprinting software, utilizing TCP/IP stack fingerprints.

Word	Definition
PRADS	Passive Real-time Asset Detection System.
Snort	An open source IDS system.
Suricata	An open source IDS system.
SYN, SYN+ACK	Synchronize, Synchronize+Acknowledge. Initial parts of the TCP handshake.

Appendix B

Source Code

This chapter of the appendix contains the source code for the program responsible for retrieving, storing and displaying asset data. This is followed by the `bridge.bro` script, which sets up the listening port in bro and defines some events useful for logging assets. Finally, I have included a script which detects Fedora installations, and records these events using the events listed in `bridge.bro`.

The python script has some requirements not native to python2: `flask`, `flask-sqlalchemy` and `broccoli-python`.

B.1 assetweb.py

```
1  from flask import Flask, render_template
2  from flask.ext.sqlalchemy import SQLAlchemy
3  from broccoli import *
4  from time import sleep
5  import json
6  from optparse import OptionParser
7  import signal, sys
8
9  app = Flask(__name__)
10 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///./assets.
       db'
11 db = SQLAlchemy(app)
12
13 class Stats():
14     def __init__(self):
15         self.os = 0
16         self.software = 0
17         self.hostnames = 0
18         self.proxy = 0
```

```
19          self.ports = 0
20          self.node = 0
21
22      def __str__(self):
23          return "[stats]: node = %i, os = %i, software = %i,
                hostnames = %i, proxy = %i, ports = %i" % (self.
                node, self.os, self.software, self.hostnames,
                self.proxy, self.ports)
24
25  stats = Stats()
26
27  class Node(db.Model):
28      id = db.Column(db.Integer, primary_key=True)
29      ip_address = db.Column(db.String(120), unique=True)
30      canonical_hostname = db.Column(db.String(120), unique=
            False)
31
32      def serialize(self):
33          retval = {}
34          retval["ip"] = self.ip_address
35          retval["canonical_hostname"] = self.
                canonical_hostname
36          retval["software"] = [x.serialize() for x in self.
                software]
37          retval["os"] = [x.serialize() for x in self.os]
38          retval["hostnames"] = [x.serialize() for x in self.
                hostnames]
39          retval["ports"] = [x.serialize() for x in self.ports
                ]
40          retval["proxynat"] = [x.serialize() for x in self.
                proxynat]
41          return retval
42
43      def __init__(self, ip, hostname=None):
44          self.ip_address = ip
45          self.canonical_hostname = hostname
46
47      def __repr__(self):
48          return '<Node %r>' % self.ip_address
49
50
51  class OS(db.Model):
52      id = db.Column(db.Integer, primary_key=True)
53      name = db.Column(db.String(120))
54      version = db.Column(db.String(120))
55      arch = db.Column(db.String(20))
56      source_script = db.Column(db.String(120))
57      unparsed = db.Column(db.Text)
```

```
58      counter = db.Column(db.Integer)
59
60      assetnode_id = db.Column(db.Integer, db.ForeignKey('node
            .id'))
61      assetnode = db.relationship('Node',
62          backref=db.backref('os', lazy='dynamic'))
63
64      def __init__(self, name, version, arch, source_script,
            unparsed, assetnode):
65          self.name = name
66          self.version = version
67          self.arch = arch
68          self.source_script = source_script
69          self.unparsed = unparsed
70          self.assetnode = assetnode
71          self.counter = 1
72
73      def serialize(self):
74          retval = {}
75          retval["name"] = self.name
76          retval["version"] = self.version
77          retval["arch"] = self.arch
78          retval["source_script"] = self.source_script
79          retval["counter"] = self.counter
80          return retval
81
82      def __repr__(self):
83          return '<OS %r>' % self.name
84
85      def __eq__(self, other):
86          return self.name == other.name and self.version ==
                other.version and self.version == other.version
87
88
89  class Software(db.Model):
90      id = db.Column(db.Integer, primary_key=True)
91      name = db.Column(db.String(120))
92      version = db.Column(db.String(120))
93      source_script = db.Column(db.String(120))
94      unparsed = db.Column(db.Text)
95      counter = db.Column(db.Integer)
96
97      assetnode_id = db.Column(db.Integer, db.ForeignKey('node
            .id'))
98      assetnode = db.relationship('Node',
99          backref=db.backref('software', lazy='dynamic'))
100
101     def __init__(self, name, version, source_script,
```

```
        unparsed, assetnode):
102         self.name = name
103         self.version = version
104         self.source_script = source_script
105         self.unparsed = unparsed
106         self.assetnode = assetnode
107         self.counter = 1
108
109     def serialize(self):
110         retval = {}
111         retval["name"] = self.name
112         retval["version"] = self.version
113         retval["source_script"] = self.source_script
114         retval["counter"] = self.counter
115         return retval
116
117     def __eq__(self, other):
118         return self.name == other.name and self.version ==
                other.version
119
120     def __repr__(self):
121         return '<Software %r>' % self.name
122
123
124 class Hostname(db.Model):
125     id = db.Column(db.Integer, primary_key=True)
126     hostname = db.Column(db.String(120))
127     source_script = db.Column(db.String(120))
128     unparsed = db.Column(db.Text)
129     counter = db.Column(db.Integer)
130
131     assetnode_id = db.Column(db.Integer, db.ForeignKey('node
            .id'))
132     assetnode = db.relationship('Node',
133         backref=db.backref('hostnames', lazy='dynamic'))
134
135     def __init__(self, hostname, source_script, unparsed,
            assetnode):
136         self.hostname = hostname
137         self.source_script = source_script
138         self.unparsed = unparsed
139         self.assetnode = assetnode
140         self.counter = 1
141
142     def serialize(self):
143         retval = {}
144         retval["hostname"] = self.hostname
145         retval["source_script"] = self.source_script
```

```
146         retval["counter"] = self.counter
147         return retval
148 
149     def __eq__(self, other):
150         return self.hostname == other.hostname and self.
                source_script == other.source_script
151 
152     def __repr__(self):
153         return '<Hostname %r>' % self.hostname
154 
155 
156 class Ports(db.Model):
157     id = db.Column(db.Integer, primary_key=True)
158     port = db.Column(db.String(20))
159     unparsed = db.Column(db.Text)
160     source_script = db.Column(db.String(120))
161     counter = db.Column(db.Integer)
162 
163     assetnode_id = db.Column(db.Integer, db.ForeignKey('node
            .id'))
164     assetnode = db.relationship('Node',
165         backref=db.backref('ports', lazy='dynamic'))
166 
167     def __init__(self, port, source_script, unparsed,
            assetnode):
168         self.port = port
169         self.source_script = source_script
170         self.unparsed = unparsed
171         self.assetnode = assetnode
172         self.counter = 1
173 
174     def serialize(self):
175         retval = {}
176         retval["port"] = self.port
177         retval["source_script"] = self.source_script
178         retval["counter"] = self.counter
179         return retval
180 
181     def __eq__(self, other):
182         return self.port == other.port
183 
184     def __repr__(self):
185         return '<Port %r>' % self.port
186 
187 class ProxyNat(db.Model):
188     id = db.Column(db.Integer, primary_key=True)
189     proxysoftware = db.Column(db.String(120))
190     unparsed = db.Column(db.Text)
```

```
191     source_script = db.Column(db.String(120))
192     counter = db.Column(db.Integer)
193 
194     assetnode_id = db.Column(db.Integer, db.ForeignKey('node
            .id'))
195     assetnode = db.relationship('Node',
196         backref=db.backref('proxynat', lazy='dynamic'))
197 
198     def __init__(self, proxysoftware, source_script,
            unparsed, assetnode):
199         self.proxysoftware = proxysoftware
200         self.source_script = source_script
201         self.unparsed = unparsed
202         self.assetnode = assetnode
203         self.counter = 1
204 
205     def serialize(self):
206         retval = {}
207         retval["proxysoftware"] = self.proxysoftware
208         retval["source_script"] = self.source_script
209         retval["counter"] = self.counter
210         return retval
211 
212     def __eq__(self, other):
213         return self.proxysoftware == other.proxysoftware
214 
215     def __repr__(self):
216         return '<ProxyNat %r>' % self.proxysoftware
217 
218 
219 def detected_node(ip_address, hostname=None):
220     node = Node(ip_address, hostname)
221     stats.node += 1
222     db.session.add(node)
223     db.session.commit()
224     return node
225 
226 def get_node(ip_address):
227     node = Node.query.filter_by(ip_address=ip_address).first
            ()
228     if node == None:
229         node = detected_node(ip_address)
230     return node
231 
232 #global detected_software: event(ip: addr, software: string,
        version: string, raw: string, source_script: string);
233 @event
234 def detected_software(ip, software, version, raw,
```

```
    source_script):
235     node = get_node(ip)
236     s = Software(software, version, source_script, raw, None
            )
237     found = False
238     for i in node.software:
239         if s == i:
240             i.counter += 1
241             found = True
242     if not found:
243         s = Software(software, version, source_script, raw,
                node)
244         stats.software += 1
245         db.session.add(s)
246         db.session.commit()
247 
248     print("%-30s \t %-30s \t %-30s \t %-30s \t %-30s" % (
            source_script, "Detected software", ip, software,
            version))
249 
250 #global detected_os: event(ip: addr, os: string, version:
        string, arch: string, raw: string, source_script: string
        );
251 @event
252 def detected_os(ip, os, version, arch, raw, source_script):
253     node = get_node(ip)
254     s = OS(os, version, arch, source_script, raw, None)
255     found = False
256     for i in node.os:
257         if s == i:
258             i.counter += 1
259             found = True
260     if not found:
261         s = OS(os, version, arch, source_script, raw, node)
262         stats.os += 1
263         db.session.add(s)
264         db.session.commit()
265     print("%-30s \t %-30s \t %-30s \t %-30s \t %-30s" % (
            source_script, "Detected OS", ip, os, version))
266 
267 #global detected_port: event(ip: addr, open_port: port, raw:
         string, source_script: string);
268 @event
269 def detected_port(ip, open_port, raw, source_script):
270     node = get_node(ip)
271     s = Ports(open_port, source_script, raw, None)
272     found = False
273     for i in node.ports:
```

```
274         if s == i:
275             i.counter += 1
276             found = True
277     if not found:
278         s = Ports(open_port, source_script, raw, node)
279         stats.ports += 1
280         db.session.add(s)
281         db.session.commit()
282     print("%-30s \t %-30s \t %-30s \t %-30s" % (
            source_script, "Detected used port", ip, open_port))
283
284 #global detected_hostname: event(ip: addr, hostname: string,
        raw: string, source_script: string);
285 @event
286 def detected_hostname(ip, hostname, raw, source_script):
287     node = get_node(ip)
288     s = Hostname(hostname, source_script, "unparsed", None)
289     found = False
290     for i in node.hostnames:
291         if s == i:
292             i.counter += 1
293             found = True
294     if not found:
295         s = Hostname(hostname, source_script, raw, node)
296         stats.hostnames += 1
297         db.session.add(s)
298         db.session.commit()
299     print("%-30s \t %-30s \t %-30s \t %-30s" % (
            source_script, "Detected responding hostname", ip,
            hostname))
300
301 #global detected_proxynat: event(ip: addr, sofware: string,
        raw: string, source_script: string);
302 @event
303 def detected_proxynat(ip, software, raw, source_script):
304     node = get_node(ip)
305     s = ProxyNat(software, source_script, raw, None)
306     found = False
307     for i in node.hostnames:
308         if s == i:
309             i.counter += 1
310             found = True
311     if not found:
312         s = ProxyNat(software, source_script, raw, node)
313         stats.proxy += 1
314         db.session.add(s)
315         db.session.commit()
316     print("%-30s \t %-30s \t %-30s \t %-30s" % (
```

```
        source_script, "Detected proxy/NAT", ip, software))
317
318 def mode_bro_bridge():
319     bc = Connection("127.0.0.1:47758")
320     print("starting")
321     while True:
322         bc.processInput()
323         sleep(0.1)
324
325 def signal_handler(signal, frame):
326     print(stats)
327     print("Exiting (ctrl+c)")
328     sys.exit(0)
329
330 @app.route("/ip/<ip>")
331 def asset_request(ip=None):
332     node = Node.query.filter_by(ip_address=ip).first()
333     if node:
334         return json.dumps(node.serialize())
335     return "{}"
336
337
338 @app.route("/")
339 def index():
340     return render_template("index.html")
341
342
343 def mode_web():
344     app.run(host="0.0.0.0")
345
346 if __name__ == "__main__":
347     signal.signal(signal.SIGINT, signal_handler)
348     parser = OptionParser()
349     parser.add_option("-m", "--mode", action="store", type="
        string", dest="mode", help="Choose mode. Either web
        or bro mode must be specified")
350     (options, args) = parser.parse_args()
351     mode = options.mode
352     print(mode)
353     if mode == "bro":
354         print("Starting in bridge mode")
355         mode_bro_bridge()
356     elif mode == "web":
357         print("Starting in web mode")
358         mode_web()
359     elif mode == "makedb":
360         print("Making initial database")
361         db.create_all()
```

```
362         db.session.commit()
363     else:
364         print("This program requires a valid mode. Try '
                assetweb.py -m web' or 'assetweb.py -m bro'")
```

B.2 bridge.bro

```
1  @load frameworks/communication/listen
2  @load base/protocols/http
3
4  # Let's make sure we use the same port no matter whether we
       use encryption or not:
5  redef Communication::listen_port = 47758/tcp;
6
7  # Redef this to T if you want to use SSL.
8  redef Communication::listen_ssl = F;
9
10 # Set the SSL certificates being used to something real if
       you are using encryption.
11 #redef ssl_ca_certificate    = "<path>/ca_cert.pem";
12 #redef ssl_private_key       = "<path>/bro.pem";
13
14 global detected_software: event(ip: addr, software: string,
       version: string, raw: string, source_script: string);
15 global detected_os: event(ip: addr, os: string, version:
       string, arch: string, raw: string, source_script: string
       );
16 global detected_port: event(ip: addr, open_port: port, raw:
       string, source_script: string);
17 global detected_hostname: event(ip: addr, hostname: string,
       raw: string, source_script: string);
18 global detected_proxynat: event(ip: addr, sofware: string,
       raw: string, source_script: string);
19
20 redef Communication::nodes += {
21         ["assetlog"] = [$host = 127.0.0.1, $connect=F, $ssl=
               F]
22 };
```

B.3 assets_fedora.bro

```
1  @load base/protocols/http
2  @load base/frameworks/software
3
4  module HTTP;
5
6
```

```
7  event http_all_headers(c: connection, is_orig: bool, hlist:
       mime_header_list)
8      {
9      local script_name = "assets_fedora.bro";
10     local asset_detection_network = 10.0.0.1/8;
11     #Ignore clients we do not want to track
12     if(c$id$orig_h !in asset_detection_network)
13         return;
14
15     if ( is_orig )
16         return; #This script only cares about information
               from server side, stopping here if communication
                is from client
17
18     if ( c?$http &&
19          c$http?$uri &&
20          c$http?$user_agent &&
21          /\/fedora\// in c$http$uri &&
22          /\/updates\// in c$http$uri &&
23          /\.d?rpm$/ in c$http$uri)
24         {
25
26         local uri = c$http$uri;
27
28         #Split url and extract variables
29         local url_vector = split_all(uri, /\//);
30         local filename = url_vector[|url_vector|];
31         local package_and_version = split(filename, /\.fc/)
               [1];
32         local package_and_version_vector = split_all(
               package_and_version, /\-/);
33         local version = cat_string_array_n(
               package_and_version_vector, |
               package_and_version_vector| - 2, |
               package_and_version_vector|);
34         local package = cat_string_array_n(
               package_and_version_vector, 1, |
               package_and_version_vector| - 4);
35
36         event detected_software(c$id$orig_h, package,
               version, uri, script_name);
37         }
38     }
```